Demo Guide

M68K

HIWARE TOOLS
Compiler
Linker
Simulator

© Copyright 1998 HIWARE

Product Manual	Manual Date
Demo Guide M68K	04/98

© Copyright 1998 HIWARE

Contents

M68K Demo Guide 5
Installation 5
Demo HIWARE TOOLS M68K Installation 5
Editor Installation 5
Win32s Installation 6
Acrobat Reader Installation 6
Overview 7
HIWARE TOOLS Demo 8
Project Directory 8
Starting with HIWARE TOOLS 8
Setting a New Project Directory 9
Editing a Project 11
Switching Between Project Directories 17
Compiler Demo 19
Switching to the Compiler Demo Project Directory 19
Starting the C Compiler 19
Setting Compiler Options 20
Selecting the Input File 22
Getting Help on a Compiler Message 23
Error Feedback in an Editor 24
Message Management 25
Linker Demo 29
Linker Parameter File 29
Linking FIBO.C 30
HI-WAVE Simulator Demo 31
Starting the Simulator 31
Pop-Up Menus in HI-WAVE 32
Loading and Starting an Application 32
Stopping an Application 33
Drag and Drop Facilities 35
View Where to Set Breakpoints 36
Working with Breakpoints 36
Stepping 38
Working with Data and Registers 39

Working with the Assembler Window 42
Simulator Specific Features. 43
Coverage Functions 43
Profiling Functions 44
Software Tracing Functions 45
Visualization Function 47
IO Stimulation. 49
How to Improve Code Efficiency 53
Introduction. 53
Compiler Options 53
SHORT Segments. 53
Defining IO Registers 54
Programming Guide Lines 55

M68K Demo Guide

Installation

Demo HIWARE TOOLS M68K Installation

To install correctly the *Demo HIWARE TOOLS M68K*, insert the disk labelled *Demo HIWARE TOOLS M68K disk1* or insert the CD-ROM in the appropriate drive. Use menu entry *Start | Run* of the windows task bar (for *Windows 95* or *NT 4.0*) or menu entry *File | Run* of the *File Manager* (for *Windows 3.11* or *NT 3.51*) to run the `SETUP.EXE` file from the disk or from the CD-ROM. The InstallShield wizard will help you to install the demo software.

By default, the installation is done in directory `C:\HICROSS`. The InstallShield wizard allows you to browse for another directory. In this document the installation path will be `C:\HICROSS`.

Editor Installation

At the end of the installation, the InstallShield wizard automatically prompts to set an editor different from *Notepad*. You can then set the editor to be used with the *Demo HIWARE TOOLS M68K*.

In this demo version, the source editor installed is *Notepad*, but any other editor could be used. Also how to set a different editor in HIWARE TOOLS is explained.

The *WinEdit* editor (shareware version) can be downloaded from the *HIWARE FTP* site:

`FTP://WWW.HIWARE.COM/PUB/HIWARE/OTHERS/WINEDIT/`

Win32s Installation

The development and debugging tools need *Win32s* to run under *windows 3.11*. If you are running this environment and if *Win32s* is not installed, you can download the *Win32s* installation from the *HIWARE FTP* site:

`FTP://WWW.HIWARE.COM/PUB/HIWARE/OTHERS/WIN32S/`

Acrobat Reader Installation

If you want to open the `.PDF` files documents and you do not have Acrobat Reader v3.01, you can download the freeware version from the *HIWARE FTP* site:

`FTP://WWW.HIWARE.COM/PUB/HIWARE/OTHERS/ACROREAD/`

If you are running *windows 3.11* download `AR16V30.EXE`

otherwise, if you are running *windows 95 or NT* download `AR32V30.EXE`.

© Copyright 1998 HIWARE

Overview

This *Demo-Guide* will help you to use the *HIWARE* development and debugging tools for the Motorola M68K microcontrollers.

HIWARE TOOLS Demo

This demo shows you how to manage your project using the *HIWARE TOOLS* shell.

Compiler Demo

This demo shows you how to compile a C source file and shows the *CM68K Compiler* features.

Linker Demo

This demo shows you how to link an application.

Simulator Demo

This demo gives an overview of the *HI-WAVE* simulator.

Simulator Specific Features

This demo gives an overview on some *Simulator* specific features.

> *Note: For all demos, the installation directory is assumed to be* `C:\HICROSS`.

HIWARE TOOLS Demo

HIWARE TOOLS is a shell delivered with the *HI-CROSS+* package. *HIWARE TOOLS* helps you to configure your development environment and provides a shortcut bar that you can fully configure to develop and debug your applications.

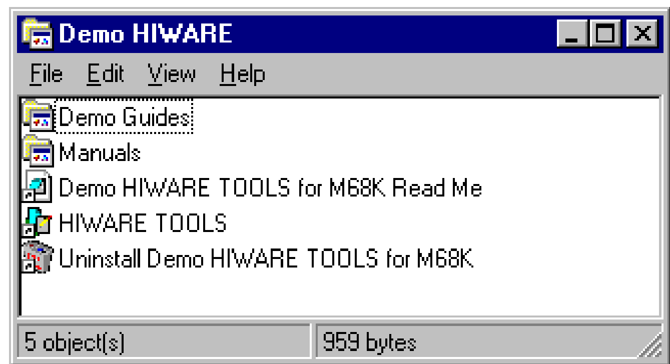
Project Directory

A project directory is a directory containing all environment files needed to configure your HIWARE development environment.

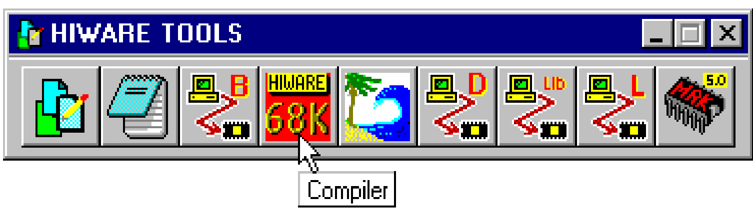
Once you have performed the installation described previously, the project directory is automatically set to the *Demo ANSI-C M68K Simulator* project directory. This directory contains initialization files that are required for the tools to work correctly. The path to this directory is: `C:\HICROSS\DEMO\SIM68KC`.

Starting with HIWARE TOOLS

If not running (if you have set an editor), you can double-click on the *HIWARE TOOLS* icon in the *Demo HIWARE* group to run the *HIWARE TOOLS* shell or select *Start* | *Program* | *Demo HIWARE* / *HIWARE TOOLS* from the Windows task bar.



The *HIWARE TOOLS* shell window pops up:



The *HIWARE TOOLS* shell window has properties identical to most of Windows

applications. A tool tip is available for each of the buttons in the *HIWARE TOOLS* shell.

When clicking one of the *HIWARE TOOLS* shell button, the corresponding executable is started and is ready to run in the specified project directory.

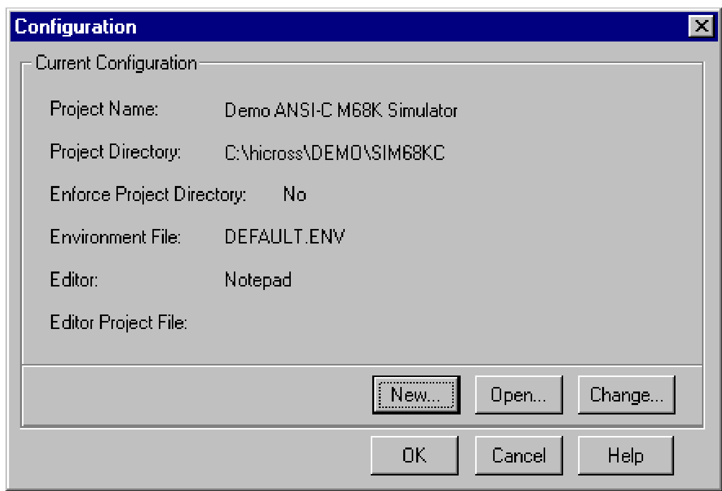
In the *HIWARE TOOLS* shell, the button with *A* stands for *Assembler*, *B* for *Burner*, *D* for *Decoder*, *Lib* for *Librarian*, *L* for *Linker* and *Mak* for *Maker*.

Click the *Compiler* button. The *HI-CROSS+ CM68K Compiler* is run and its main window is displayed. The Compiler environment is correctly set to compile files from the project directory. Close the *Compiler* window.

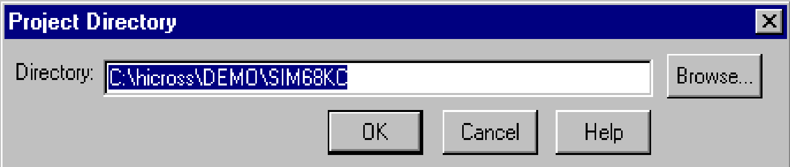
Setting a New Project Directory

In order to write your own applications, you may need to set another project directory. This can be done using the *HIWARE TOOLS* shell. For that purpose, follow these steps:

1. When clicking the first icon of the bar, you open the *Configuration* dialog which enables to set your environment through different dialogs. Click to open the *Configuration* dialog.

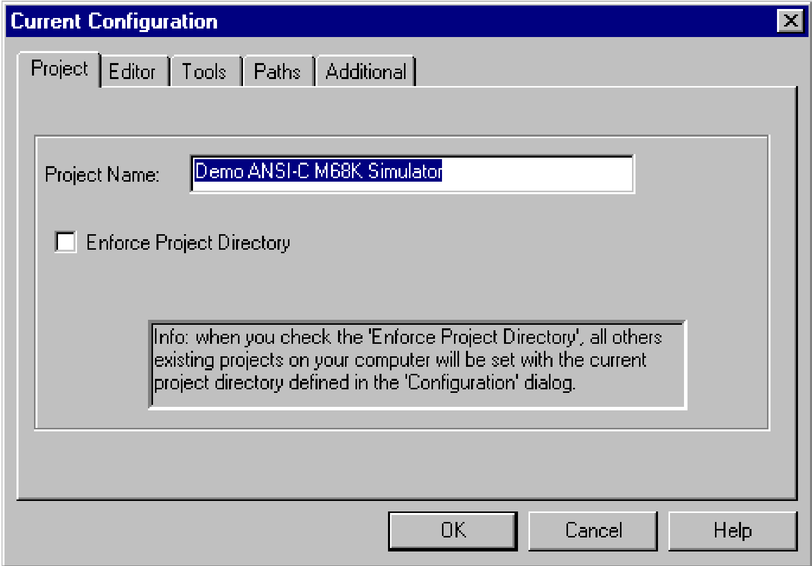


2. Click *New*. The *Project Directory* dialog pops up.

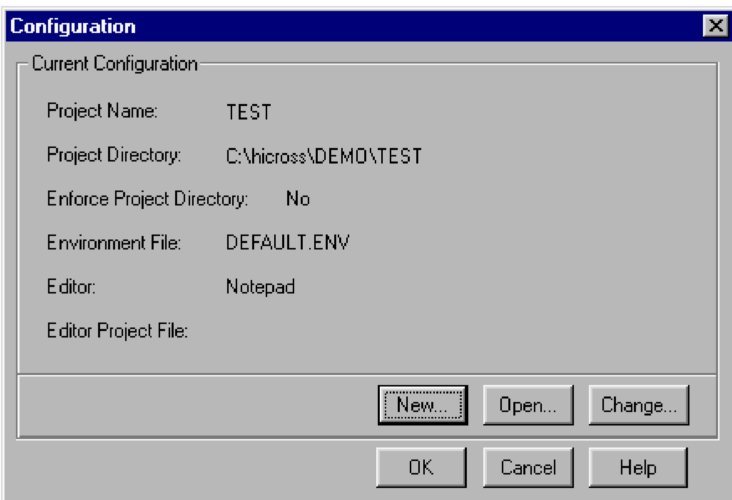


You can type in the edit box or browse for the directory path to set a new project directory. If this directory does not already exists, it is created on your hard disk (*Note that only one level of unexisting directory can be created (no subdirectory!)*).

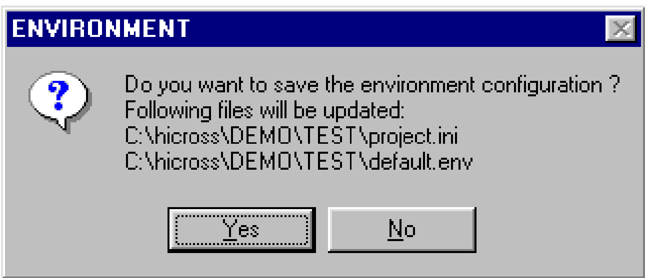
3. As an example type in `C:\HICROSS\DEMO\TEST`. When clicking *OK*, the *New Configuration* dialog is displayed. All the parameters of your project can be set, starting by the *Project Name* as shown below.



4. Change the *Project Name* from *Demo ANSI-C M68K Simulator* to “TEST”. Then you can progress from index to index and set all the parameters of your project. All the fields are initialized by default with the settings of the previous project directory.
5. When you click *OK*, the new project is recorded. In the *Configuration* dialog, the *Project Name* and *Project Directory* fields change: the *Project Name* is now `TEST` and the *Project Directory* is `C:\HICROSS\DEMO\TEST`.



6. Click *Ok* and the *ENVIRONMENT* dialog is opened and asks if you want to save the settings.

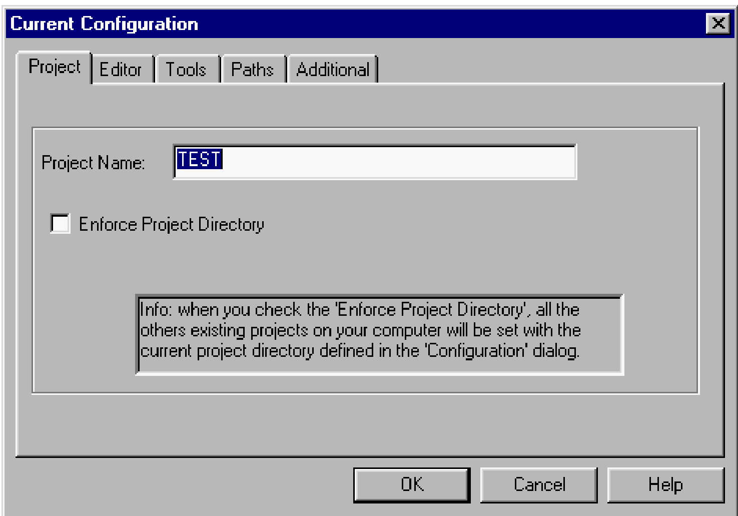


7. Click *Yes* in the *ENVIRONMENT* dialog to update the configuration files. The dialog is closed and the *HIWARE TOOLS* shell is set to run in the new project directory. All the needed initialization files have been stored in this new directory.

Editing a Project

Each project has its own properties and you can set all the tools and properties associated to your project with the *HIWARE TOOLS* shell.

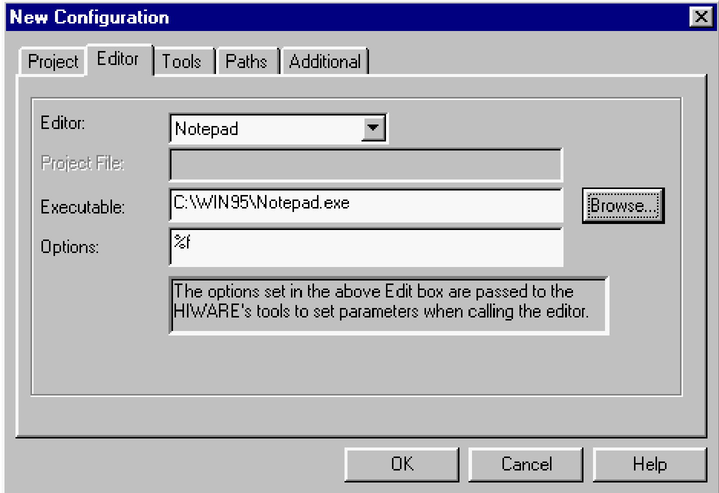
From the *Configuration* dialog, click *Change...* to edit the current project and reach the following dialogs.



Setting another Editor

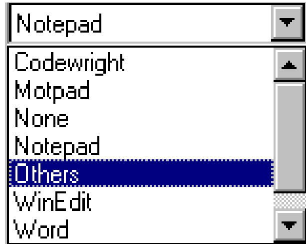
When installing the *Demo HIWARE TOOLS M68K*, the source editor is set by default to *Notepad*. If you want to use another editor, it must be set in the *HIWARE TOOLS* shell as it is described below.

1. Select the index *Editor* in the *Current Configuration* dialog. The settings for *Notepad* are displayed in the index.



2. In the *Editor* combo box you can select one of the commonly used editor with the *HIWARE TOOLS* shell: *WinEdit*, *CodeWright* or a different Windows text editor.

© Copyright 1998 HIWARE



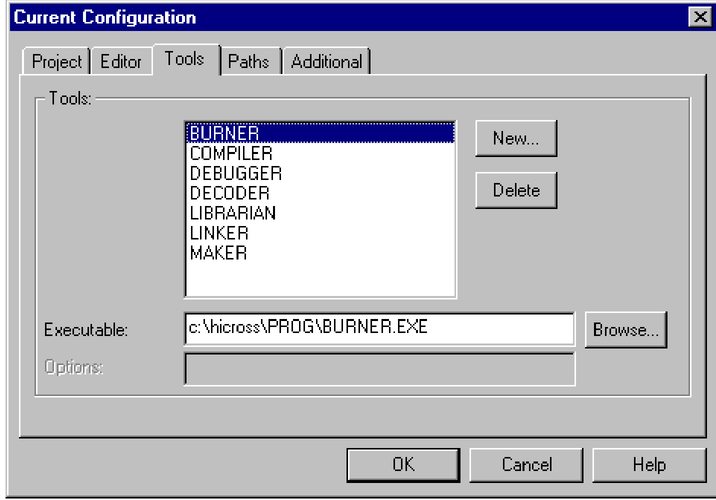
3. Select your future editor in the combo box. If you set *WinEdit* or *Codewright*, the *Project File* edit box is enabled to enter an editor project configuration file, respectively a `.WPJ` or `.PJT` file.
4. The *Options* edit box is automatically updated with `%f, /#:%l`, etc. These parameters are used for automatic error feedback to open the source file and source line when double-clicking an error report line in the *Compiler* or the *Assembler* window.

Note: The content of the ***Options*** *edit box is appended to the command line specified in the Executable edit box, when the editor is started by a HIWARE tool.* **`%f`** *is a modifier referring to a file name.* **`%l`** *can be used to refer to a line number.*
Notepad cannot be started with a line number as parameter, the modifier **`%l`** *does not appear in the Options edit box.*

5. Then browse for the executable file. Click *Browse*. The *Search Editor* dialog is displayed. Select the future editor `.EXE` file in its installation directory and click *Open*.
6. The *Current Configuration* dialog is updated with the new editor path.
7. Click *OK* to close this dialog. The *Editor* field in the *Configuration* dialog is now set with the new editor name.
8. Click *OK* to close the *Configuration* dialog. Click *Yes* in the *ENVIRONMENT* dialog to update the configuration files and the *HIWARE TOOLS* shell icon.

Setting the Tools in HIWARE TOOLS Shell

Tools can be added or removed from the *HIWARE TOOLS* shell. This allows you to customize the *HIWARE TOOLS* shell and to add your own tools and utility programs in the toolbar. In the *Configuration* dialog, click *Change...* to open the *Current Configuration* dialog. Select the index *Tools*.

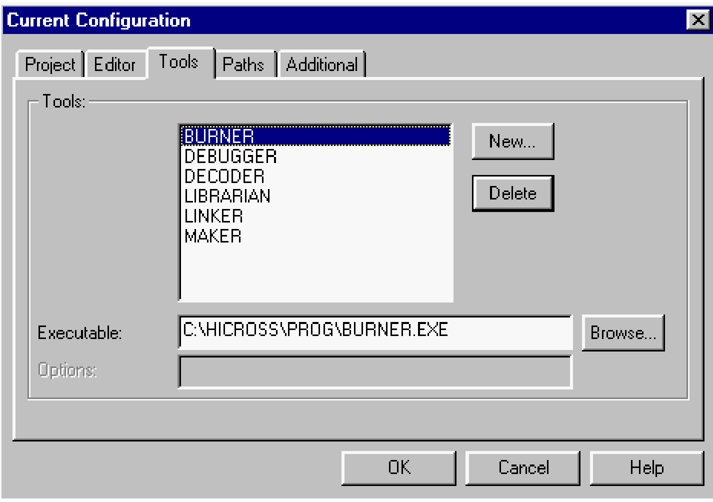


Applications visible in the shell toolbar are set in this dialog. When clicking on one of the names displayed in the listbox, the corresponding application executable (including path) is displayed in the *Executable* edit box.

Removing a Tool

A tool can be easily removed from the *HIWARE TOOLS* shell. As an example, we will remove the *CM68K Compiler* from the toolbar.

1. Select `COMPILER` in the listbox and click *Delete*. The `COMPILER` tool is removed from the *Tools* listbox.



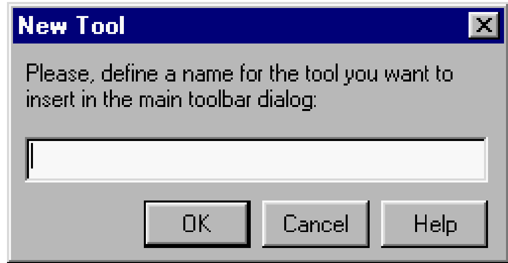
© Copyright 1998 HIWARE

2. Click *OK* to close the *Current Configuration* dialog.
3. Click *OK* to close the *Configuration* dialog. Click *Yes* in the *ENVIRONMENT* dialog to update the configuration files. The *Compiler* icon has disappeared from the *HIWARE TOOLS* shell.

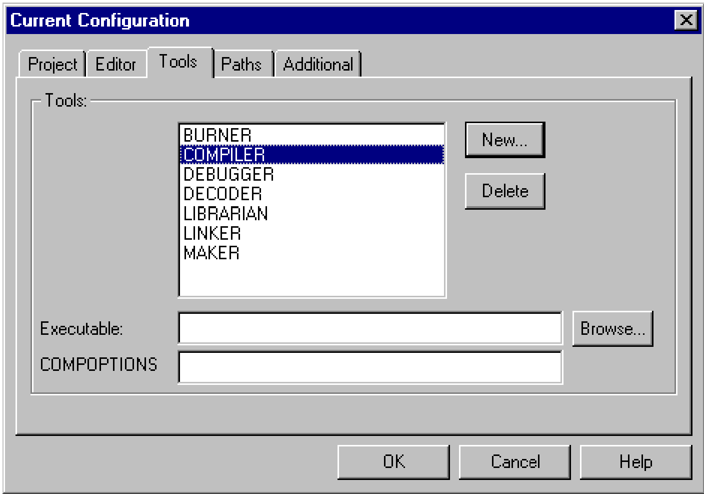
Adding a Tool

To set the Compiler back in the *HIWARE TOOLS* shell, follow these steps:

1. Click to open the *Configuration* dialog and click *Change...* to open the *Current Configuration* dialog. Select the index *Tools*.
2. Click *New...* the *New Tool* dialog is displayed.

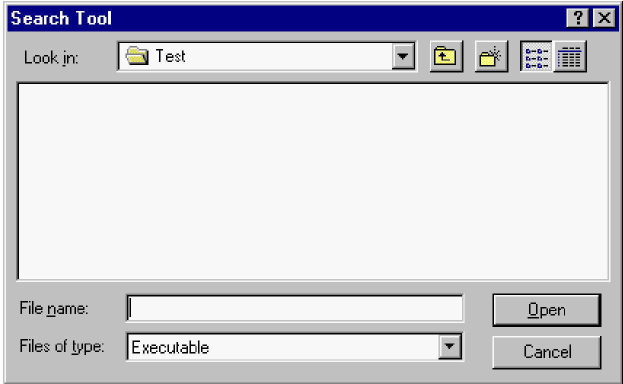


3. Type “COMPILER” and click *OK*. This name will be used for the tool tip in the shortcut bar.
4. `COMPILER` is added to the list of available tools in the listbox.

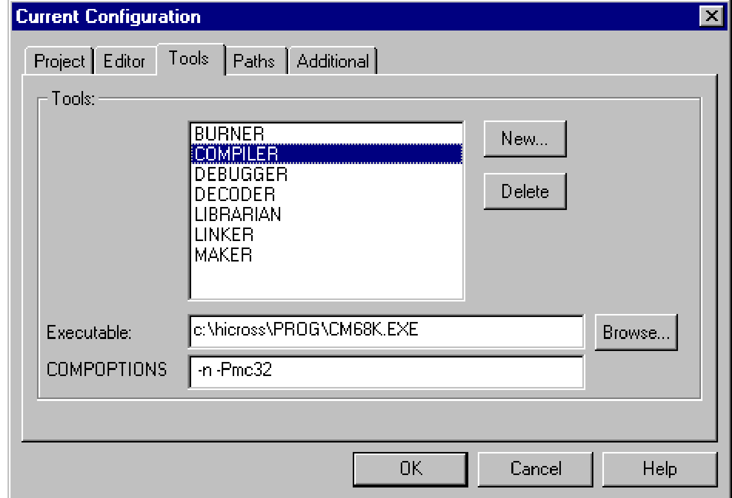


5. An executable has to be associated with the newly defined tools. This is done in the *Executable* edit box. The path for the executable can be directly entered or

browsed. To associate the executable file to the *COMPILER* item, select *COMPILER* in the *Tools* listbox and click *Browse...* . The *Search Tool* dialog is displayed.



6. Use this dialog to browse for the *HI-CROSS+ CM68K Compiler*. The command line should be `C:\HICROSS\PROG\CM68K.EXE`. When the correct executable has been set, click *Open* to close the *Search Tool* dialog. This path is displayed in the *Executable* edit box.



7. Set also the default compilation options in the *COMPOPTIONS* edit box.
8. Click *OK* to close the *Current Configuration* dialog.
9. Click *OK* to close the *Configuration* dialog. Click *Yes* in the *ENVIRONMENT* dialog to update the configuration files. The *HIWARE TOOLS* shell is displayed. The *HI-CROSS+ CM68K Compiler* icon is back in the toolbar.

10.Click on this icon to start the Compiler. Choose the menu entry *File | Exit* to quit

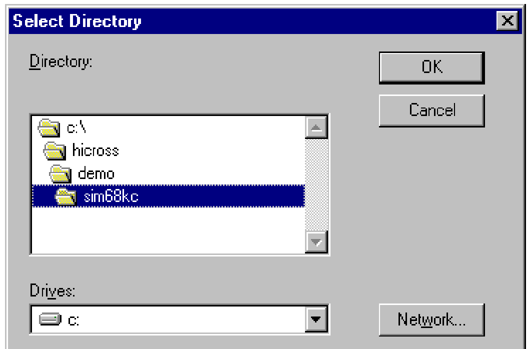
the Compiler.

If the *HIWARE TOOLS* shell cannot find an application defined in the *Executable* edit box, the following icon is displayed in the short-cut bar:

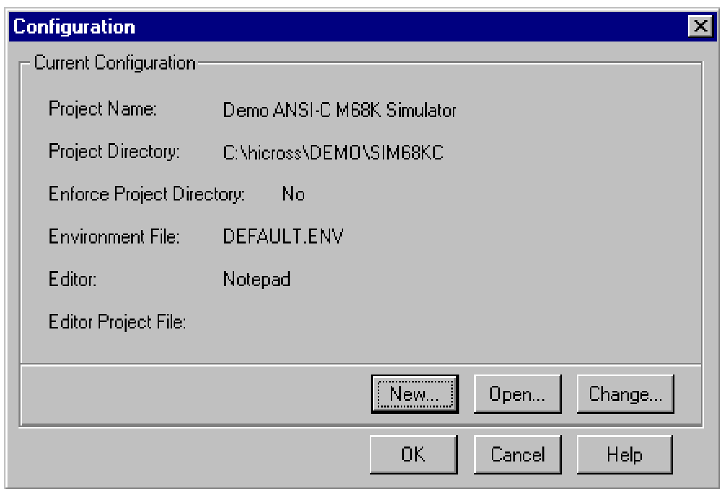
Switching Between Project Directories

To switch back from the TEST project directory to the Demo ANSI-C M68K Simulator project directory (to open another project directory), follow these steps:

1. In the *Configuration* dialog, click *Open...* The *Select Directory* dialog is opened. Browse for the directory containing the Demo ANSI-C M68K Simulator demo. The path should be C:\HICROSS\DEMO\SIM68KC.



2. Click *OK*. The *Select Directory* dialog is closed.



3. In the *Configuration* dialog, the *Project Name*, *Project Directory* and *Editor* fields change: the *Project Name* is now Demo ANSI-C M68K Simulator, the *Project Directory* C:\HICROSS\DEMO\SIM68KC and the *Editor* is

`Notepad`. Click *OK* to close the *Configuration* dialog. The *HIWARE TOOLS* shell is ready to work in the *Demo ANSI-C M68K Simulator* project directory.

Compiler Demo

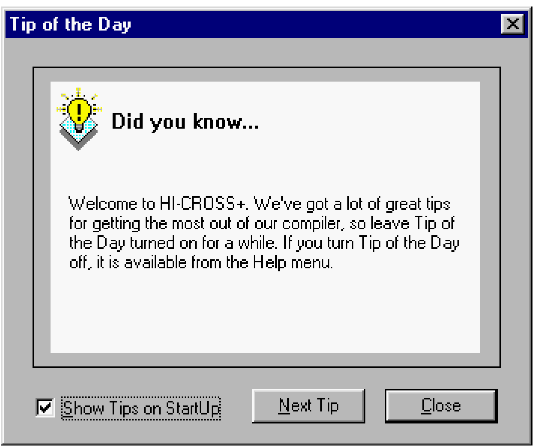
Switching to the Compiler Demo Project Directory

To perform the Compiler demo, use the *HIWARE TOOLS* shell to switch to the `Demo ANSI-C M68K Simulator` project directory. The path to this directory is: `C:\HICROSS\DEMO\SIM68KC`.

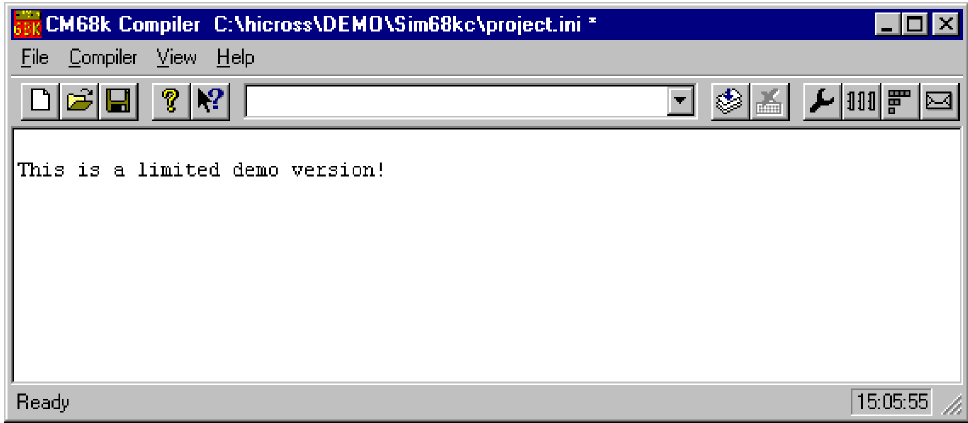
Starting the C Compiler

Start the *HIWARE TOOLS* shell and click the *Compiler* button to start the C Compiler.

The *Tip of the Day* dialog box is opened.

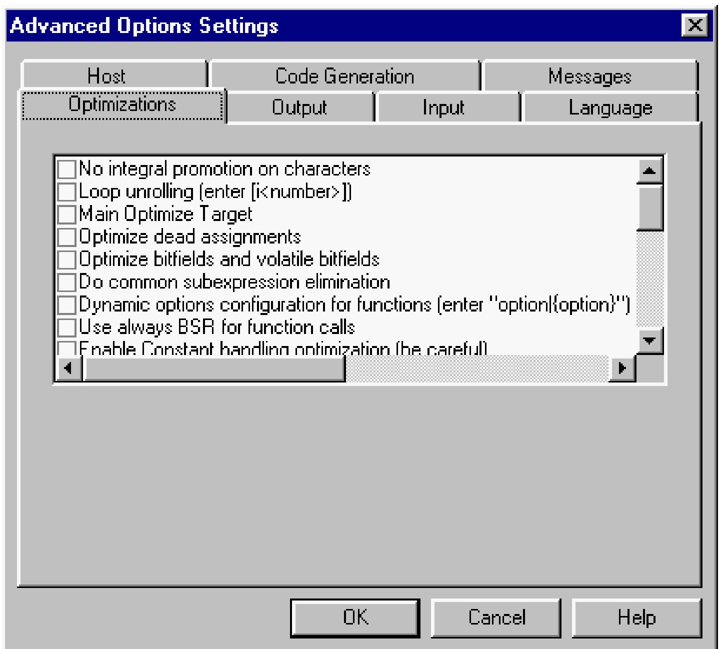


Click *Close* to close this dialog box. The main *HI-CROSS+ Compiler* window is displayed.



Setting Compiler Options

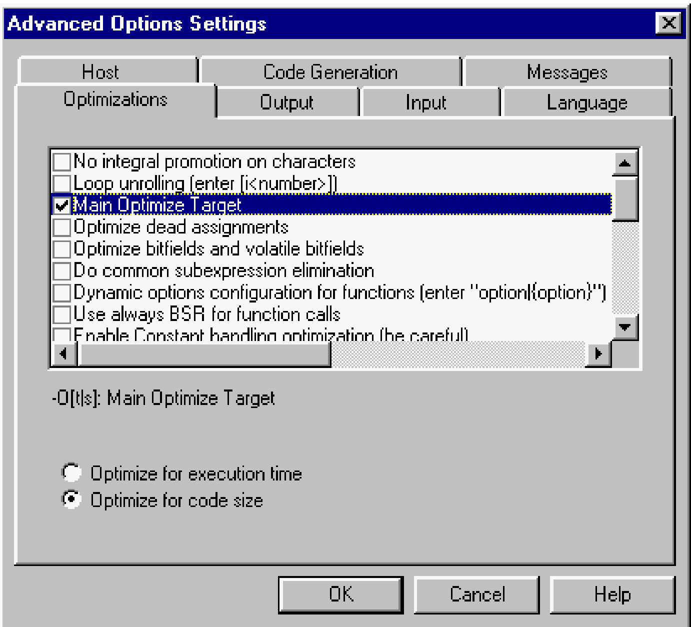
Compiler options can be set or reset choosing *Compiler | Options | Advanced* or clicking . The *Advanced Options Settings* dialog is opened.



Different indexes are available for Compiler option settings. To set a Compiler option, check the box in front of the desired option. More information could be required and set in the lower part of the index page. When the correct settings have been specified, close the dialog using *OK* to set the changes.

> *Note: About compiler option settings, please see also section* ***How to Improve Code Efficiency*** *at the end of this demo guide.*

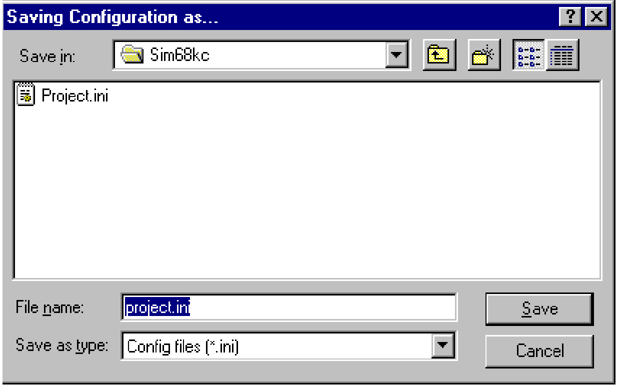
For further option details, please refer to the *Compiler* manual.



To obtain *Help* on an option, select this option an press the F1 key. The corresponding topic of the *HelpFile* is automatically opened.

Those settings may be saved using the menu entry *File | Save Configuration as...* .

The *Saving Configuration as...* dialog is entered.



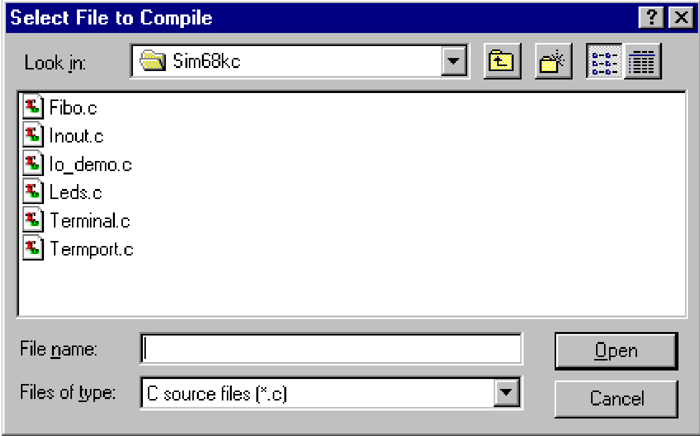
You can specify a file name for your configuration file in the edit box (e.g. `MyConfig.INI`) and click *Save*.

By default, you can save these options in your current project configuration file: `PROJECT.INI`.

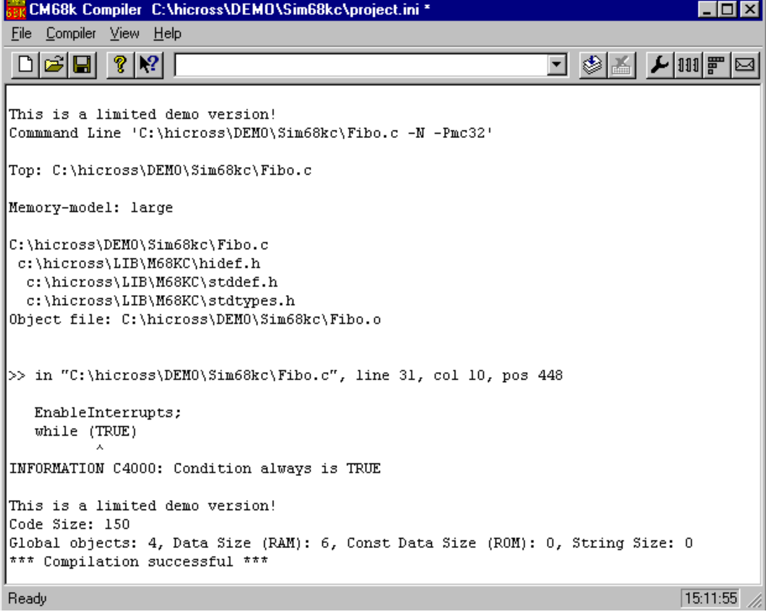
Selecting the Input File

To select the input file which must be compiled you have the following possibilities:

- Select *File | Compile*. The *File to Compile* dialog box is opened. Select the file you want to compile, `C:\HICROSS\DEMO\SIM68KC\FIBO.C` and click *Open*. The file is automatically compiled.



- Start the *Explorer* or the *File manager* and switch to the directory `C:\HICROSS\DEMO\SIM68KC`. Select the file `FIBO.C` and drag it to the *Compiler* window. The file is compiled as soon as you drop it.
- Another way to compile a file is to use the edit box and the associated list from the *Compiler* window toolbar. Type in directly the name of the file to compile in the edit box or select it from the listbox and click in the toolbar or press Enter to compile the file.



Getting Help on a Compiler Message

Create an Error Test File

Using *Notepad*, open `FIBO.C` in the `C:\HICROSS\DEMO\SIM68KC` directory.

In the main of the program, put in comment variable i declaration: `// int i;`

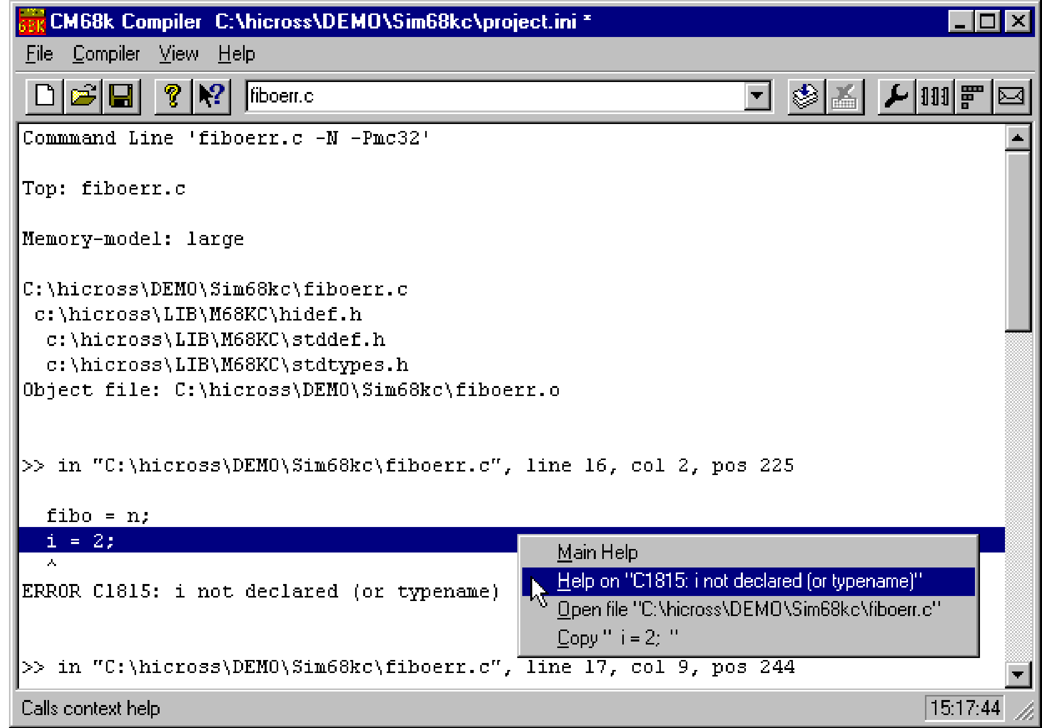
Save the file as `FIBOERR.C` and close the editor.

Getting Help using the F1 Key

- Once the file `FIBOERR.C` has been compiled, errors are generated.
- Click one of the lines where the source line generating the error message is displayed and press F1. The help file is opened and shows a description of the error message.
- Press Esc to close the help file.

Getting Help using the Right Mouse Button

- Once the file `FIBOERR.C` has been compiled, errors are generated.
- Points to the code line generating the error message and right-click. A *Context Menu* pops up.



- This menu contains four entries:
  - When selecting the *Main Help* entry, the Main help file topic is loaded.
  - When selecting *Help on ...* the help file is opened and shows a description of the error message.
  - When selecting *Open File ...* the source file is loaded in the specified editor (at the moment *Notepad*). The `FIBOERR.C` file is opened in *Notepad.* If your editor understands the "`%l`" command line parameter (that you can set in the *Editor* index in the HIWARE TOOLS configuration), the cursor reaches automatically the line producing the error.
  - The last entry of the pops up menu *Copy ...* is to be used for *Error Feedback* with editors that do not allow for a line number as parameter.

Error Feedback in an Editor

Error Feedback with Editors Accepting a Line Number as Parameter (%l)

When compiling the file `FIBOERR.C`, errors are issued. Error Feedback can be done using two ways:

1. Double-click one of the line where the source line generating the error message is displayed.:
   - The editor is started and the source file is automatically opened.
   - The cursor reaches the line where the error message has been issued.
   - Close the editor.
2. Using the *Context Help* pop up menu:
   - Points to the line generating the error and right-click.
   - The *Context Help* pop up menu is displayed. Select *Open File ...*
   - The source file is loaded in the editor. The cursor reaches the line where the error message has been issued.
   - Close the editor.

Error Feedback with Editors which Do Not Accept Line Number as Parameter: Notepad

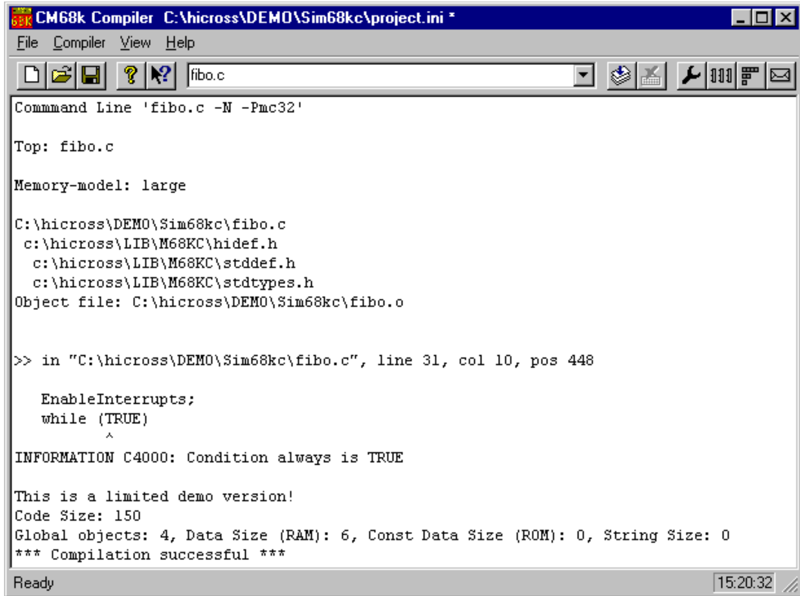
Compile the file `FIBOERR.C`, errors are issued. Error feedback can be done in two ways:

1. Click in the *Compiler* window one of the source line generating an error message:
   - Press Ctrl + C to copy the current line in the clipboard.
   - Double-click on this line the *Notepad* is started and the source file is automatically opened.
   - Select *Search | Find*, the *Find* dialog box is opened.
   - Press Ctrl + V to copy the content of the clipboard in the edit box, then click *Find next*. *Notepad* will jump to the line, where the error message has been issued (in another example, it is possible that the first occurrence of the researched string is not the one which generates the error).
   - Close *Notepad*.
2. Using the *Context Help* pop up menu:
   - Right-click in the *Compiler* window on a source line generating an error message.
   - The *Context Help* pop up menu is displayed. select *Copy ...* to copy the current line in the clipboard.
   - Right-click in the *Compiler* window on the source line again.
   - In the *Context Help* pop up menu entry *Open ...* the *Notepad* is started and the source file is automatically opened.
   - Select *Search | Find*, the *Find* dialog box is opened.
   - Press Ctrl + V to copy the content of the clipboard in the edit box, then click *Find next*. *Notepad* will jump to the line, where the error message has been issued (in another example, it is possible that the first occurrence of the researched string is not the one which generates the error).
   - Close *Notepad*.

Message Management

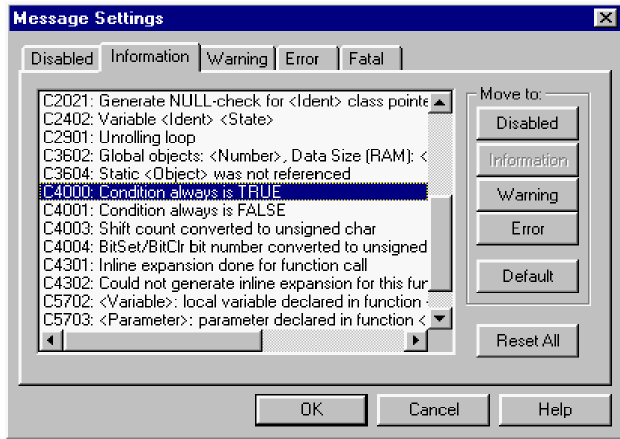
When compiling a file, the Compiler may issue a message. There are four classes of messages *Fatal*, *Error*, *Warning*, *Information*. A message can also be *Disabled*.

When compiling `FIBO.C`, an *Information* Message is issued:



Disabling an Information Message

- Click or choose *Compiler | Messages*. The *Message Settings* dialog is entered.
- This dialog allows to switch the message class. Select `C4000: Condition always is TRUE` message in the *Information* index. This is the message generated when compiling `FIBO.C`.

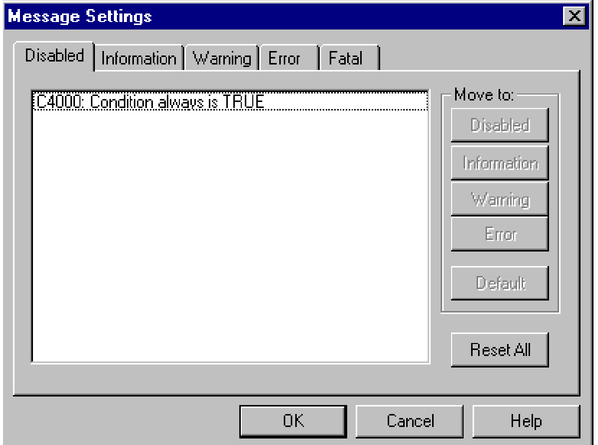


- Click the *Disabled* button. This dialog pops up:

- Click *Yes*. The `C4000` message disappears from the list of *Information* mes-

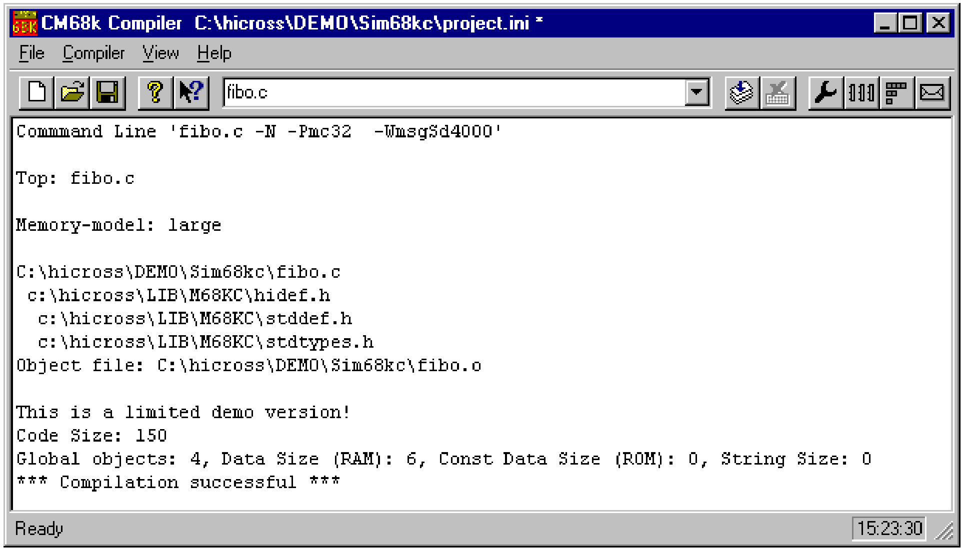
sages.

- Select the *Disabled* index. The C4000 message is displayed here. This index contains all the disabled messages.



- Click *OK* in the *Message Settings* dialog.

When compiling FIBO.C, no information message is generated anymore.



Moving Back the Information Message

To enable the C4000 information message back:

- Open the *Message Settings* dialog and select the C4000 message in the *Disabled* index.
- Click the *Information* button from the *Move to:* button group. The Move selected messages dialog pops up.
- Click *Yes*. The C4000 information message is removed from the *Disabled* index.
- Select the *Information* Folder. The C4000 message can be retrieved in the infor-

mation messages list.

- Click *OK* in the *Message Settings* dialog.
- Choose *File | Exit* to close the *HI-CROSS+ Compiler*.

Linker Demo

Once a file has been assembled or compiled it needs to be linked so that the Simulator or a debugger can run it. This demo should be performed in the C:\HICROSS\DEMO\SIM68KC project directory.

Linker Parameter File

The input file for the Linker is a parameter file. The extension used is .PRM .

As an example FIBO.PRM, the parameter file matching the FIBO.C file compiled previously, looks like:

```
LINK fibo.abs

NAMES  fibo.o startmc.o ansi.lib
END
SECTIONS
    MY_RAM = READ_WRITE  0x4000 TO 0x43FF;
    MY_ROM = READ_ONLY   0x1000 TO 0x3FFF;
PLACEMENT
    DEFAULT_ROM, ROM_VAR, STRINGS  INTO  MY_ROM;
    DEFAULT_RAM                    INTO  MY_RAM;
END
VECTOR ADDRESS 0x04 _Startup /* set reset vector on _Startup */
STACKSIZE  0x200
```

LINK fibo.abs Name of the generated executable.

The executable file produced by the Linker will be called FIBO.ABS.

NAMES ... END Names of the object files involved in the application.

The file FIBO.ABS uses procedures or global variables from the files FIBO.O, STARTMC.O and ANSI.LIB.

SECTIONS Memory Map: defines RAM and ROM areas.

Sections MY_RAM, MY_ROM are defined.

PLACEMENT ... END Assignment of the different sections in the memory.

The code of the file FIBO.ABS, the strings and constant variables will be placed into section MY_ROM. The global data of FIBO.ABS will be placed into section MY_RAM.

VECTOR Initialise reset vector.

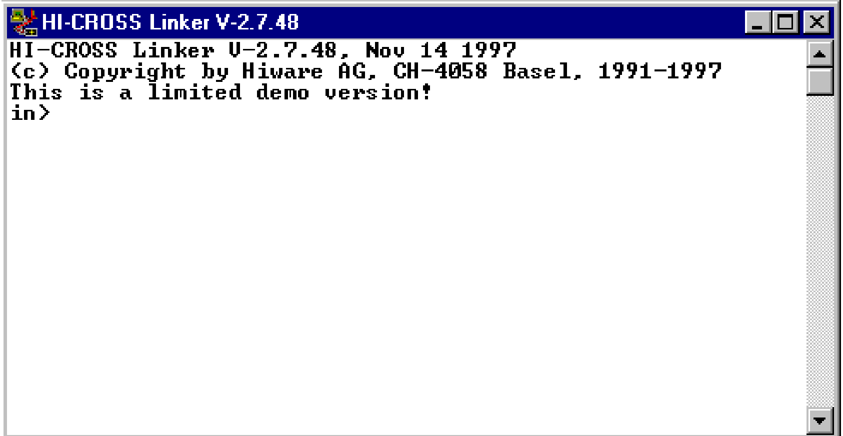
STACKSIZE Initialise the number of bytes to reserve for the stack.

For more information on the parameter files, please refer to *Linker* documentation.

Linking FIBO.C

- To start the *Linker*, click the corresponding button in the *HIWARE TOOLS* shell.

- The *Linker* window is opened and you are prompted for a Linker Parameter File name.



- On the command line type `FIBO.PRM` and press Enter to link the file.
- The *Linker* displays the following information in the main window:

```
HI-CROSS Linker V-2.7.48
HI-CROSS Linker V-2.7.48, Nov 14 1997
(c) Copyright by Hiware AG, CH-4058 Basel, 1991-1997
This is a limited demo version!
in>fibo.prm
reading directories of:
C:\HICROSS\DEMO\SIM68KC\fibo.o
c:\hicross\LIB\M68KC\startmc.o
c:\hicross\LIB\M68KC\ansi.lib
Searching objects ...
Allocating objects ...
Making Fixups ...
Writing EMap ...
listing of linkprocess to C:\HICROSS\DEMO\SIM68KC\fib
o.MAP

Executable to: c:\hicross\DEMO\SIM68KC\fibo.abs

 ****** linking succeeded ******


in>|
```

- Linking is successful. Press Esc to close the *HI-CROSS Linker* window.

HI-WAVE Simulator Demo

HI-WAVE is a Multipurpose Tool that can be used for various tasks in the embedded system and industrial control world. Some typical tasks are:

- Simulation and debugging of an embedded application.
- Simulation and/or cross-debugging of an embedded application.
- Simulation of a hardware design (e.g., board, processor, I/O chip).
- Building a target application using an object oriented approach.
- Building a host application controlling a plant using an object oriented approach.

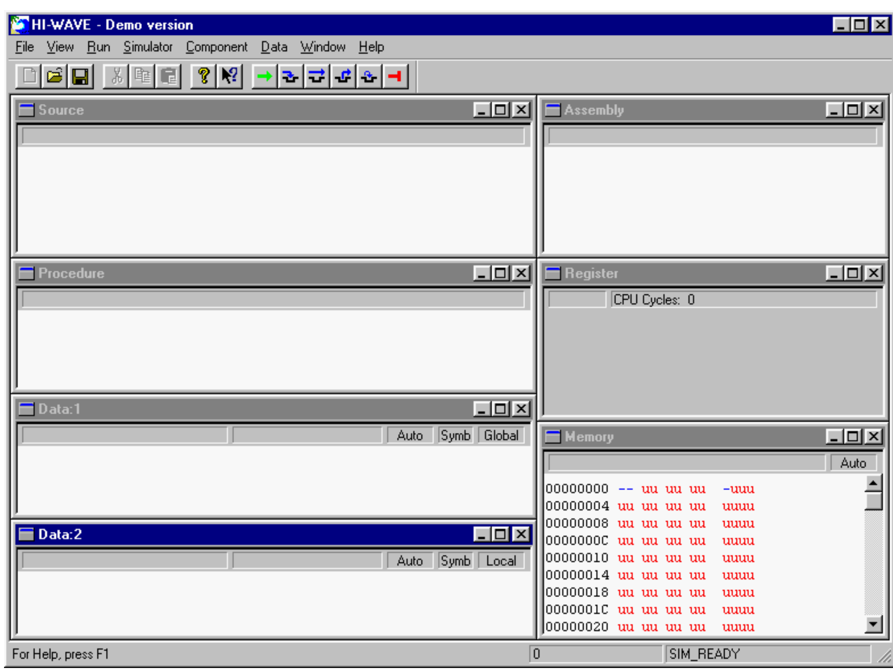
The actions described in this chapter should be performed using the simulator (*Sim* target).

Starting the Simulator

To run this demo using the *Simulator*, switch to the project directory `C:\HICROSS\DEMO\SIM68KC` using the *HIWARE TOOLS* shell.

Click the *HI-WAVE* button in the *HIWARE TOOLS* shell.

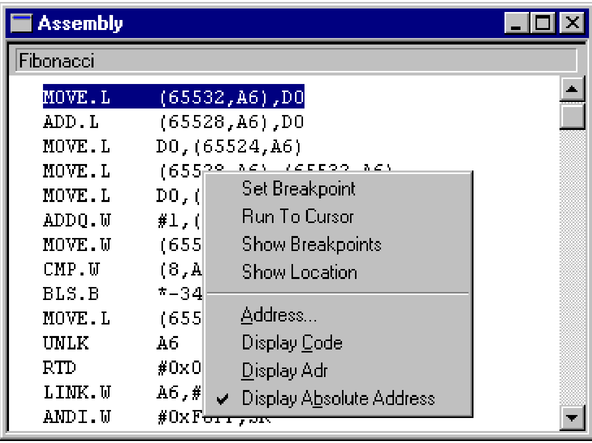
The application is started and the simulator target is set.



SIM_READY indicates in the status line that the simulator is ready.

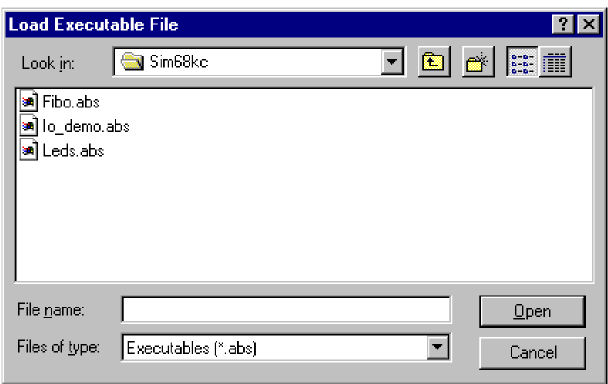
Pop-Up Menus in HI-WAVE

Right-clicking over a component window (e.g. *Assembly* component window) opens the corresponding pop-up (context sensitive) menu, containing the functions associated with the component.

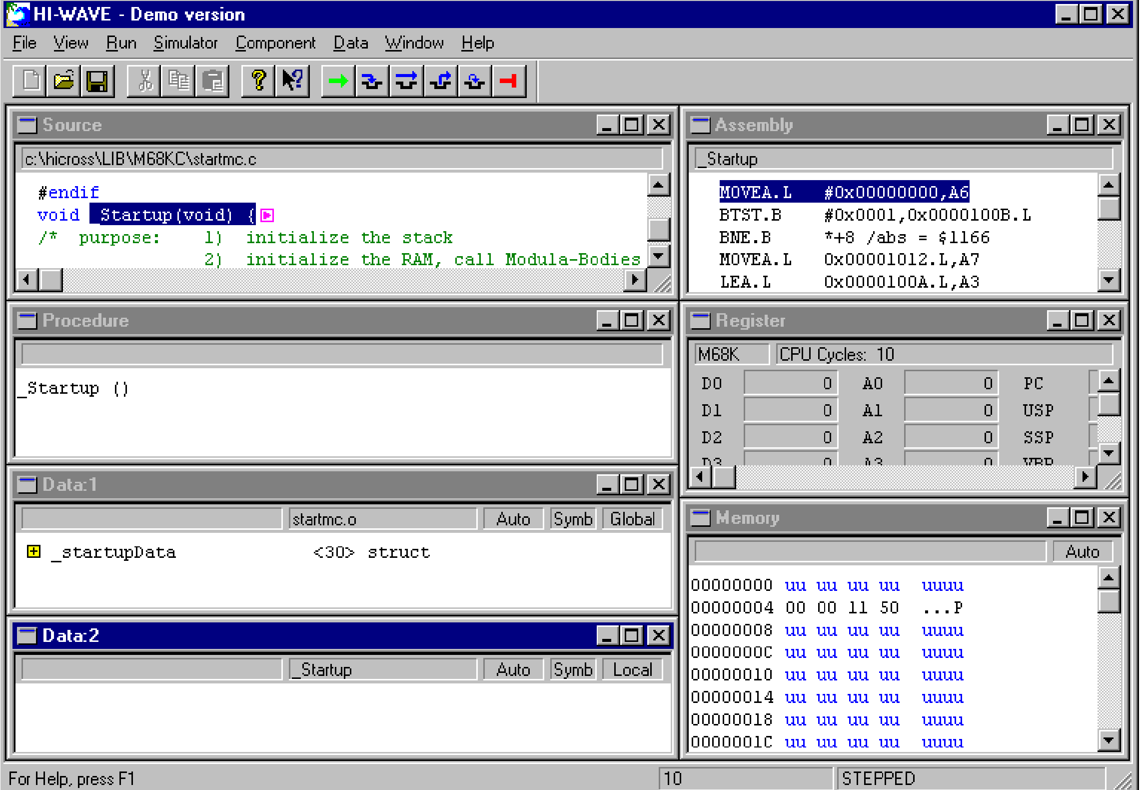


Loading and Starting an Application

1. Choose *Simulator | Load*. The *Load Object File* dialog is then opened.



2. Select the file `FIBO.ABS` and click *Open*.
3. The dialog is closed and the program is loaded.
4. You can click twice the *Single Step* icon to pass the prestart code and display the STARTMC.C startup file in the *Source* window.



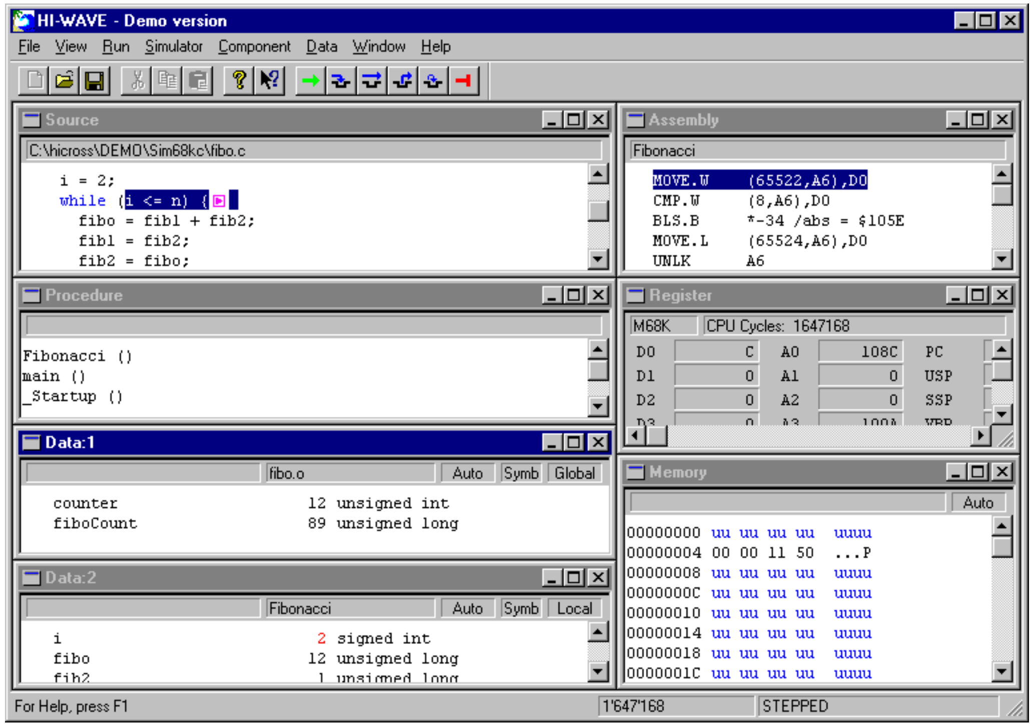
5. Choose *Run* | *Start/Continue* or click .
6. The application is started.

RUNNING Indicates in the status line that the application is running.

Stopping an Application

1. Choose *Run* | *Halt* or click . The execution of the program is stopped.

The highlighted line The blue highlighted line in the *Source* window is the statement on which the program was stopped. (i.e. the next statement that will be executed.) The blue highlighted line in the *Assembly* window is the assembly instruction on which the program was stopped. (i.e. the next assembly instruction that will be executed.)



HALTED Indicates in the status line that the application was stopped.

Fibonacci This procedure is displayed on the first line of the *Procedure* window. It is the procedure currently being executed.

main This procedure is displayed on the second line of the *Procedure* window. It is the procedure which has invoked the function `Fibonacci`.

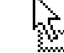
Global In the object info bar (top line of a component window) of the *Data:1* component indicates that global variables of module `FIBO` are displayed below.

counter **12** Indicates that the current value of variable `counter` is 12.

Local In the object info bar (top line of a component window) of the *Data:2* component indicates that local variables are displayed in this window.

Fibonacci In the object info bar (top line of a component window) of the *Data:2* component indicates that local variables of `Fibonacci` are displayed in the window.

n **12** Indicates that the current value of variable `n` is 12.

Drag and Drop Facilities 

How to Drag and Drop an Object

1. Select the component containing the object you want to drag.
2. Make sure the destination component where you want to drag the object is visible.
3. Select the object you want to drag.
4. Press and hold down the left mouse button, drag the object onto the destination component and then release the mouse button.

Drag and Drop Combinations

Drags and drops of objects are possible between different component windows. See below the most interesting possible combinations of drag and drop between components and associated actions. Other combinations are possible, please refer to the *HI-WAVE* manual.

Drag and Drop from Data to Memory

- Drag and drop `fiboCount` from the *Data:1* window to the *Memory* window.
- The memory where `fiboCount` is located is dumped. Its location is high-lighted in the *Memory* component.

Drag and Drop from Register to Memory

- Point to the `PC` register in the *Register* window.
- Drag and drop it in the *Memory* window. The memory starting at the address pointed by `PC` is dumped. The corresponding address is high-lighted in the *Memory* component.

Drag and Drop from Procedure to Data:2 (Local)

- Select `main()` or `Fibonacci` in the *Procedure* window.
- Drag and drop it in the *Data:2* window. All the local variables of the dragged procedure are displayed in the *Data:2* window.

Drag and Drop from Procedure to Source

- Select `main()` or `Fibonacci` in the *Procedure* window.

- Drag and drop it in the *Source* window. The source corresponding to the dragged procedure is displayed.

View Where to Set Breakpoints

When programming in High Level Programming Language, it may be difficult to determine where breakpoints can be set. Especially when the code is optimized. HI-WAVE allows you to display marks in front of each position in your source code, where a breakpoint can be set.

1. Click onto the title bar of the *Source* window. The *Source* window is activated and the *Source* menu appears in the menu bar.
2. Choose *Source | Marks*. All the statements where a breakpoint can be set are identified by a special mark:

`fib2 = fibo;` Indicates in the *Source* window that a breakpoint can be set at this statement.

3. Choose *Source | Marks*. All marks are then removed.

Working with Breakpoints

Once the application is loaded, HI-WAVE allows you to control the execution of your application and to stop this execution under certain conditions using break-points. There are four kinds of breakpoints:

temporary breakpoints The application will be stopped the next time it tries to execute the statement. This breakpoint is deleted as soon as it is reached.

permanent breakpoints The application will be stopped each time it reaches the statement. This breakpoint remains valid until the user explicitly deletes it.

counting breakpoints The application will be stopped once it has executed the statement a given number of times.

conditional breakpoints The application will be stopped at the statement as soon as a given condition is true.

© Copyright 1998 HIWARE

Setting Temporary Breakpoints

1. Point to statement `fib2 = fibo;` in the *Source* window,
2. Right-click and choose *Run To Cursor* in the pull down menu. The program continues execution and stops before executing the statement.

Breakpoint In the status line indicates that the program was stopped due to a breakpoint.

3. Choose *Run | Start/Continue* or click . The program continues execution and won't stop any more.

Setting Permanent Breakpoints

1. Choose *Run | Halt* or click . The program is then stopped.
2. Point to the statement `return(fibo);` in the *Source* window.
3. Right-click and choose *Set Breakpoint* in the pull down menu.

This symbol in the *Source* window points to the statement in which the breakpoint is defined.

4. Choose *Run | Start/Continue* or click . The application continues execution and stops before executing the selected statement.

Breakpoint In the status line indicates that the program was stopped due to a breakpoint.

In our demo program, local variable `n` is incremented each time the program enters the function `Fibonacci`. Please note the current value of this variable.

5. Choose *Run | Start/Continue* or click . The program continues execution and is stopped again the next time it reaches the breakpoint.

If you compare the current value of local variable `n` with the previous one, you will see that the variable `n` was incremented by 1. The function `Fibonacci` has been executed once again.

6. Point again to statement `return(fibo);` in the *Source* window.
7. Right-click and choose *Delete Breakpoint* in the pull down menu. The permanent breakpoint in the application is deleted.

Stepping

HI-WAVE also supports different stepping functions, to allow a better control over your application execution. There are four kinds of stepping actions:

single step The application will be stopped at the next source statement.

step over The application will be stopped at the next source statement, but function calls are skipped.

step out The application will be stopped at the next source statement in the caller function.

assembly step The application will be stopped at the next assembly statement.

Single Step on C Level

Choose *Run | Single Step* or click . The application stops at the next high-level language statement. If the application is currently stopped on a function call, a *Single Step* stops the application on the first instruction in the called function.

Step Over

Click . The application stops at the next high-level language statement of the current function. If the application is currently stopped on a function call, a *Step Over* stops the application on the instruction following the function call.

Step Out

Click . If the application was previously stopped in a function (subroutine), a *Step Out* stops the application on the source instruction following directly the function call. If you step out on the top level of your application, HI-WAVE is then “RUNNING”, as no function call is found.

Single Step on Assembler Level

Choose *Run | Assembly Step* or click . The application stops at the next assembler statement.

© Copyright 1998 HIWARE

Working with Data and Registers

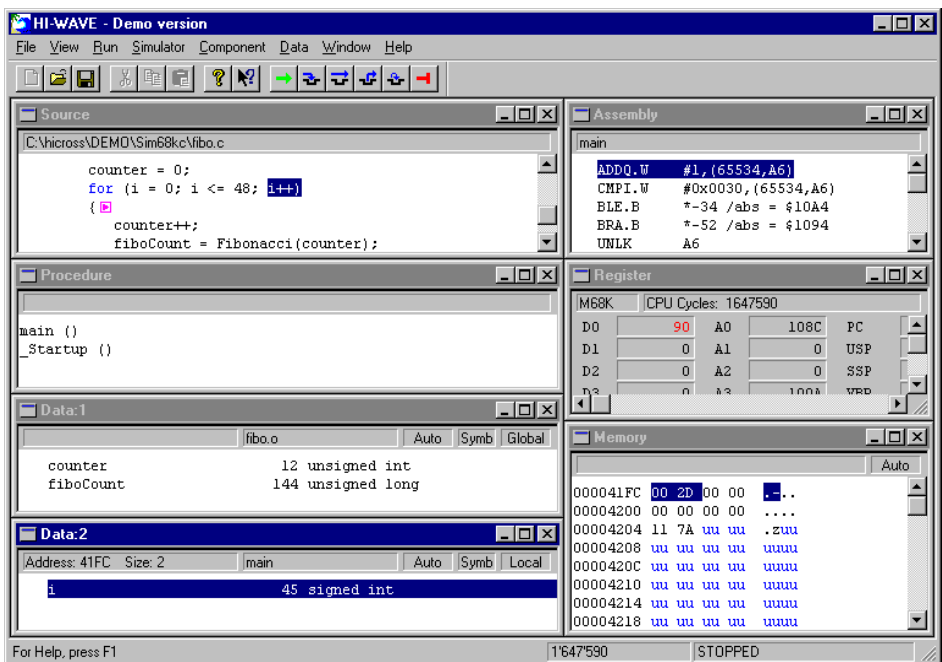
The features of the *Data*, *Register* and *Memory* windows are described below.

Displaying and Updating Local Variables

Local variables of procedures that are listed in the *Procedure* window can be displayed and eventually updated. (Note: this feature can only be used when the application is not running).

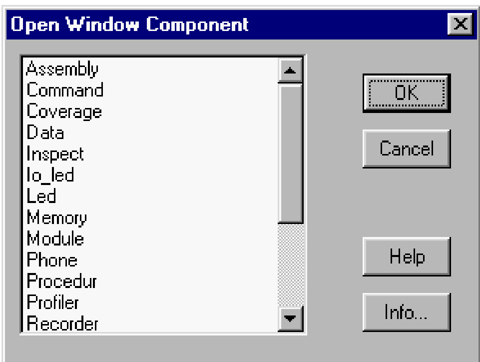
Local In the object info bar (top line of a component window) of the *Data:2* component indicates that local variables are displayed.

1. Double-click on variable `i` in the *Data:2* component.
2. Now you can change the value of this variable. Type “45” and press Enter .
3. The value displayed next to variable `i` is now 45. This value has directly been updated in the target system's RAM if you are working on a target board.

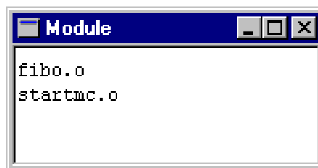


Displaying and Updating Global Variables

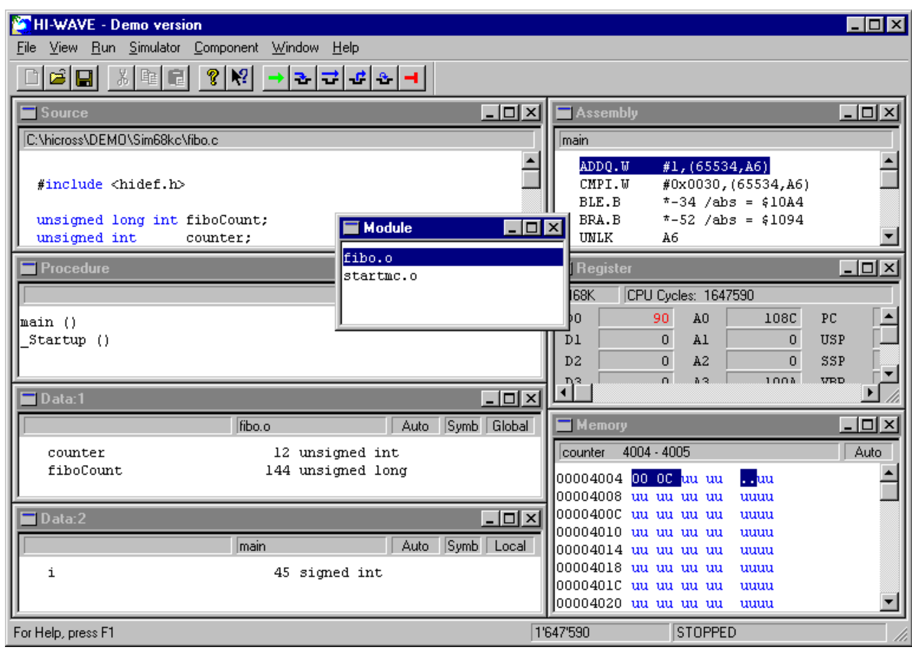
1. Choose *Component | Open...* . The *Open Window Component* dialog pops up.



2. Select *Module* and click *OK*. The *Module* window is now opened.



3. Global variables of modules that are listed in the *Module* window can be displayed and updated. In the *Module* window: double-click on `fibo.o`.



Global Indicates in the heading line of the *Data:1* component that global variables are displayed.

4. Double-click on the variable `counter` in the *Data:1* window. You can change the value of this variable.

5. Press Esc : the value of the variable `counter` remains unchanged.

Drag and Drop from *Module* component to *Data:1* (Global) component.

- Select `STARTMC.o` in the *Module* component.
- Drag and drop it in the *Data:1* window. The global variables from the selected module are displayed in the *Data:1* window.

Drag and Drop from *Module* component to *Source* component.

- Select `STARTMC.o` in the *Module* component.
- Drag and drop it in the *Source* window. The source corresponding to the selected module is displayed in the *Source* window.

Displaying and Updating Register Contents

The *Register* window displays the current values of all the CPU registers.

Point to register `D0` in the *Register* window and double-click.

You can change the `D0` register value (be careful, before updating any registers, you have to be sure that the new register contents do not disturb the application).

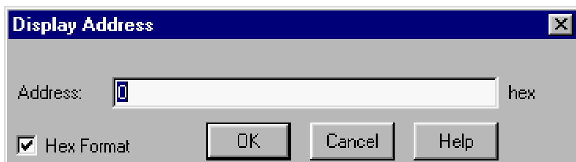
Press Esc to restore the previous value of the register.

Inspecting Memory

In the *Memory* window, current content of the simulated target system's memory starting at address `$00` is displayed in hexadecimal format.

The start address for global data defined in the Linker Parameter File `FIBO.PRM`. is address `$4000`. The data should then be located starting at this location.

1. Click the title bar of the *Memory* window. The *Memory* window is activated and the *Memory* menu appears in the menu bar.
2. Choose *Memory | Address...* . The *Display Address* dialog is opened.



3. Here you can change the base address for the memory display. Enter `4000`. This location is reached in the *Memory* component. With the mouse cursor, point and click on memory hexdumps in the *Memory* component. When existing, associated symbol names, size and memory locations are displayed in the object info bar (top line of the component window) while the mouse button is pressed.

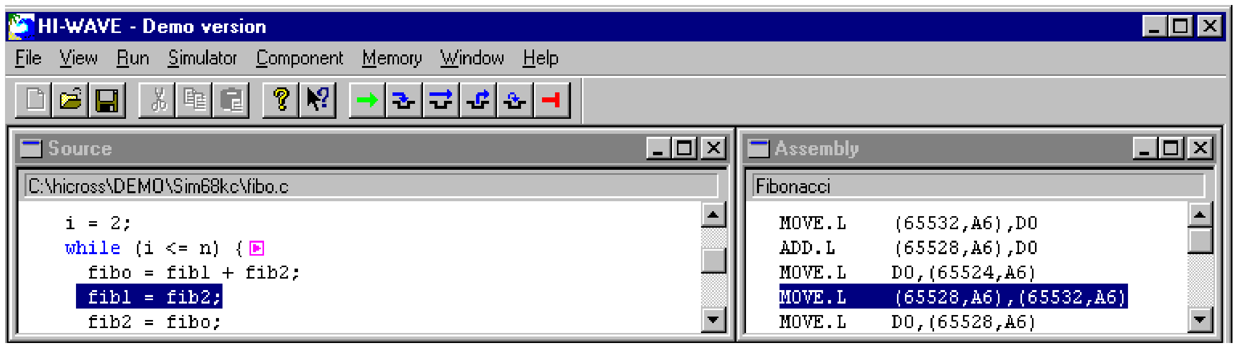
4. Choose *Memory | Format | UDec*. Memory contents are shown in unsigned decimal format.
5. Other data formats are also available. Choose *Memory | Format | Hex* to reset the format to hexadecimal.

Working with the Assembler Window

On-line Disassembling

While the application is stopped you can display assembler statements associated with high-level language statements.

1. Double-click on `Fibonacci()` in the *Procedure* component.
2. Point the mouse before the statement `fib1 = fib2;` and select with the mouse this expression (click and hold down while moving to the right)
3. Click on the selection with the mouse and drag it to the *Assembly* window. The drag and drop cursor appears on the screen.
4. When the *Assembly* window is reached, drop the selection. The assembler statements corresponding to the selected high-level statement are greyed.



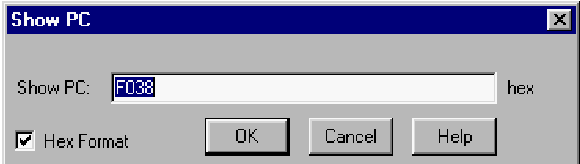
grayed lines The grayed lines in the *Assembly* window are assembler statements generated by the Compiler for the selected high-level language statement.

Assembler statements are disassembled directly from the simulated target system's ROM (or RAM). As a consequence, when, for example, the code has been destroyed due to an addressing error, it is immediately visible.

Consulting the Code

1. Click on the title bar of the *Assembly* window. The *Assembly* window is activated and the *Assembly* menu appears in the HI-WAVE main menu.

2. Choose *Assembly | Display Code*. Hexadecimal code associated with each assembler statement is displayed to the left of the statement.
3. Choose *Assembly | Display Adr*. The memory location in the simulated target system's ROM (or RAM) of each assembler statement is displayed.
4. Choose *Assembly | Display Code* from the menu again. The hexadecimal code is removed.
5. Choose *Assembly | Display Adr*. The memory locations are removed.
6. Choose *Assembly | Address...* . The *Show PC* dialog is opened.



7. In this dialog you can define an address from where you want to have the code displayed.
8. Click *Cancel* to close this dialog.

Working with Breakpoints in the Assembly Window

Breakpoints can be set in the *Assembly* window in the same way as they are set in the *Source* window.

Simulator Specific Features

Use the *HIWARE TOOLS* shell to switch to the `C:\HICROSS\DEMO\SIM68KC` project directory.

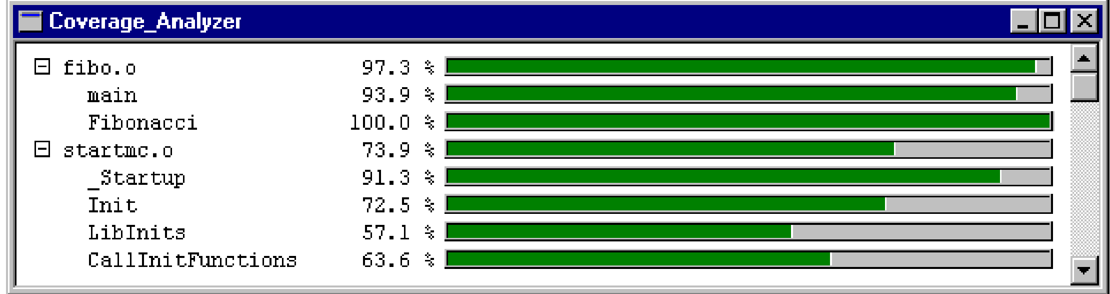
Coverage Functions

The HI-WAVE simulator provides coverage analysis information using the *Coverage_Analyzer* component. This component displays the percentage of code executed in each module and function of the application currently executed.

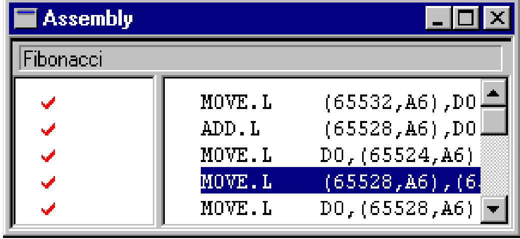
A Source Code Coverage analysis can be introduced through the `FIBO.ABS` application:

1. Choose *File | Exit* to quit *HI-WAVE*.
2. Start *HI-WAVE* using the *Debugger* button in the *HIWARE TOOLS* shell.

3. Choose *Simulator | Load* and load `FIBO.ABS`. Click to start the execution of the application.
4. After a short while, click to stop the application.
5. Choose *Component | Open... .* The *Open Window Component* dialog pops up.
6. Select *Coverage* in the listbox and click *OK*. The *Coverage Analyzer* component pops up. Coverage information about the different modules in the application are displayed in the component.



7. Click . Coverage information about the different functions in the module are displayed.
8. Drag from the *Coverage* component to the *Source* or *Assembly* component. Coverage information about the source or assembly instructions are displayed in the destination component.

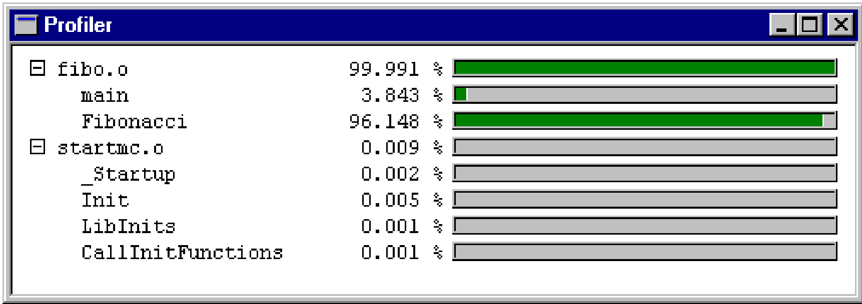


The symbol in front of a statement means that it has been executed.

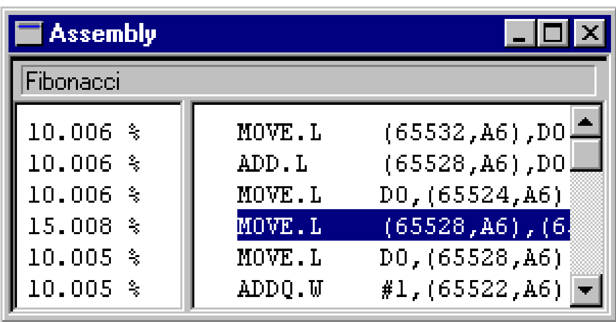
Profiling Functions

The *Profiler* component allows to determine how long (in percentage) the application spends in each module, function or statement.

1. Choose *Component | Open... .* The *Open Window Component* dialog pops up.
2. Select *Profiler* in the listbox and click *OK*. Profiling information about the different modules in the application are displayed in the component.



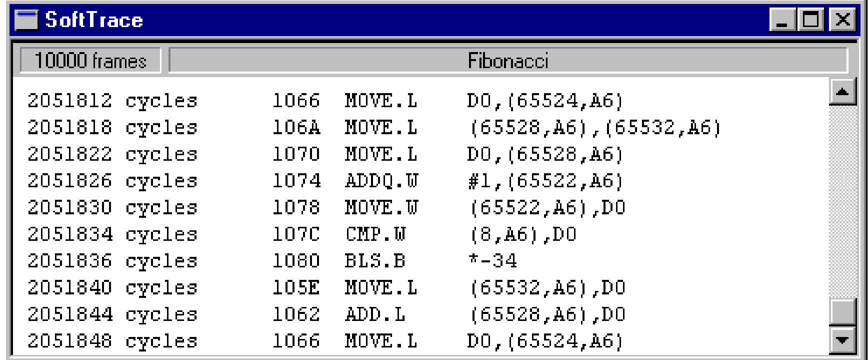
3. Click in front of a module name: profiling information about the different functions in the module are displayed in the component.
4. Drag from the *Profiler* component into the *Source* or *Assembly* component. Profiling information are displayed in the destination component.



Software Tracing Functions

The *SoftTrace* component records the latest cycles executed in the application. This component gives you an overview over the program flow.

1. Choose *File | Exit* to quit *HI-WAVE*.
2. Start *HI-WAVE* using the *Debugger* button in the *HIWARE TOOLS* shell.
3. Choose *Simulator | Load* and load `FIBO.ABS`.
4. Choose *Component | Open...* . The *Open Window Component* dialog pops up.
5. Select *Softtrac* in the listbox and click *OK*. The *SoftTrace* component window pops up.
6. Click to start the execution of the application.
7. After a short while, click to stop the execution.



8. The recorded frames are displayed in the *SoftTrace* component.

Interaction between the Source, the Assembly and the Trace Components

The *SoftTrace* component offer the possibility to retrieve in the Source and Assembly components the instruction corresponding to a frame.

Scrolling the SoftTrace component

1. Move slightly the *SoftTrace* component window in order to see also the *Source* and *Assembly* components.
2. Use the scroll bars in the *Trace* component to find the code you want to retrieve in the *Assembly* and *Source* components.

Points the mouse cursor to the frame you want the associated code and click on it. When sliding up and down the selected frame (keep pressing the left mouse button), the corresponding code is grayed in the *Source* and the *Assembly* components.

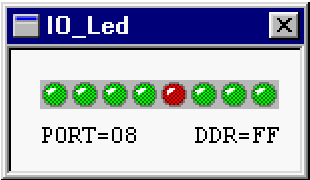
Visualization Function

The HI-WAVE simulator also support IO visualization components. The present installation provides two visualization components, but it is possible to implement your own visualization component using the ***IO Simulation Development Kit***.

IO_LED Component

The `LEDS.ABS` demo program switches the different leds in the *IO_LED* component. The leds are the digital binary representation from the content of memory at address `0x210`.

1. Choose *File | Exit* to quit *HI-WAVE*.
2. Start *HI-WAVE* using the *Debugger* button in the *HIWARE TOOLS* shell.
3. Choose *Simulator | Load* and load `LEDS.ABS`.
4. Choose *Component | Open...* . The *Open Window Component* dialog pops up.
5. Select *IO_LED* in the listbox and click *OK*. The *IO_LED* component window pops up.
6. Click to start the execution of the application. The leds are switched ON/OFF.



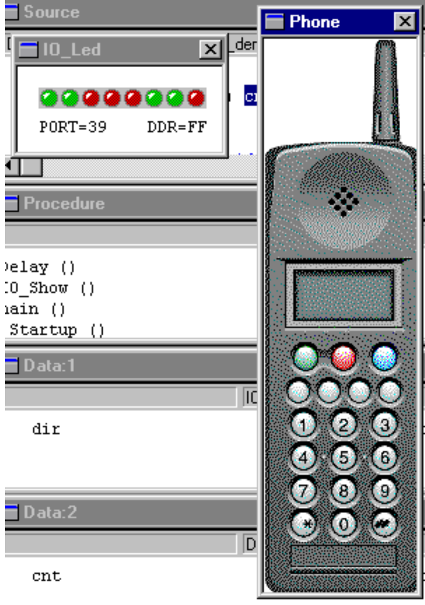
Telephone Component

The HI-WAVE simulator also allows you to simulate your final product. This allows you to test your software before the product is available. This can be implemented using the IO Simulation Development Kit.

`LEDS.ABS` demo program offers the possibility to switch ON/OFF the different leds in the *IO_LED* component when dialing on the *PHONE* component.

1. Choose *File | Exit* to quit *HI-WAVE*.
2. Start *HI-WAVE* using the *Debugger* button in the *HIWARE TOOLS* shell.
3. Choose *Simulator | Load* and load `LEDS.ABS`.
4. Choose *Component | Open...* . The *Open Window Component* dialog pops up.

5. Select *IO_LED* in the listbox and click *OK*. The *IO_LED* component window pops up.
6. Using the same procedure, open the *Phone* component.
7. Click to start the execution of the application. After a while, click to stop.



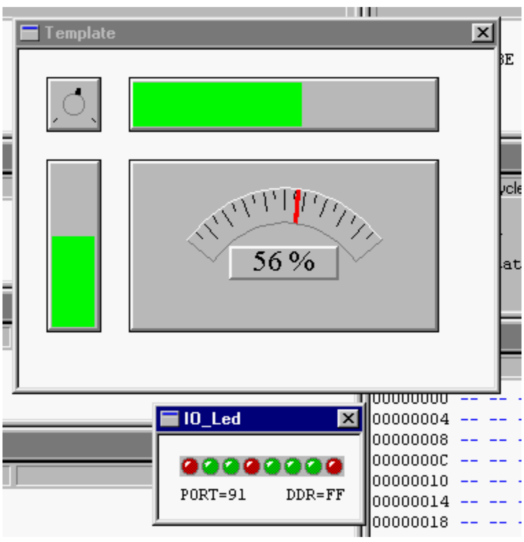
8. Click one of the button of the phone. The leds in *IO_LED* displays the ASCII code corresponding to the value of the phone button you have selected.

Template Component

`IO_DEMO.ABS` demo program switches the different leds in the *IO_LED* component and also changes the display in the *Template* Component.

The *Template* Component is the analog representation from the content of memory at address `0x210`.

1. Choose *File | Exit* to quit *HI-WAVE*.
2. Start *HI-WAVE* using the *Debugger* button in the *HIWARE TOOLS* shell.
3. Choose *Simulator | Load* and load `IO_DEMO.ABS`.
4. Choose *Component | Open...* . The *Open Window Component* dialog pops up.
5. Select *IO_LED* in the listbox and click *OK*. The *IO_LED* component window pops up.
6. Using the same procedure, open the *Template* component.
7. Click to start the execution of the application.



8. The leds are switched ON/OFF and the content of the *Template* component is periodically changed.

IO Stimulation

HI-WAVE also supports ***IO Stimulation***. Using this feature you can generate (stimulate) interrupts or memory access generated by an external IO device.

Stimulated Memory Access

Stimulation File

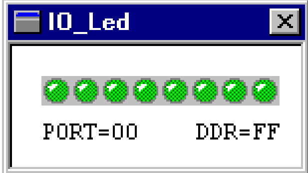
Using an editor, open the file named `IO_VAR.TXT` located in directory `E:\HICROSS\DEMO\SIM68KC`. This file looks as follows:

```
/* Define an identifier a, which is located at address 0x210*/
/* This identifier is 1 Byte wide.*/
def a = TargetObject.#210.B;
/* After 200 000 cycles have expired, repeat 50 time */
/* the code sequence specified between the keywords */
/* PERIODICAl and END. */
PERIODICAL 200000, 50:
     50000 a = 128; /* After 50 000 cycles, write 128 at address 0x210. */
    150000 a = 4;   /* After 150 000 cycles, write 4 at address 0x210. */
END
10000000 a = 0; /* After 10 000 000 cycles, write 0 at address 0x210. */
```

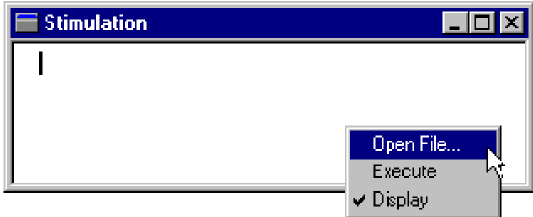
First, the stimulated object is defined. This object is located at address 0x210 and is 1 byte wide. Once 200’000 cycles have been executed, the memory location 0x210 is accessed periodically 50 times. First the memory location is set to 128 and then 100’000 cycles latter, it is set to 4.

Running stimulation

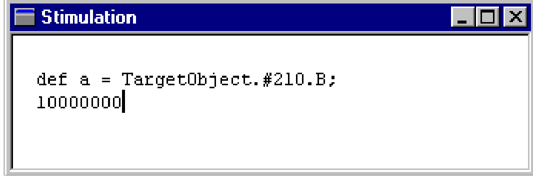
1. Start *HI-WAVE* using the *Debugger* button in the *HIWARE TOOLS* shell.
2. If no target is set, choose *Component | Set Target* and load the `Sim` (simulator).
3. Choose *Component | Open* and load the *IO_LED* component.
4. Click with the left mouse button on each led in the IO_LED component. The leds are then enabled (displayed green).



5. Choose *Component | Open* and load the *Stimulat* component.
6. Click on the *Stimulation* window to activate it.
7. Choose *Stimulation* | *Open File* and open `IO_VAR.TXT`.

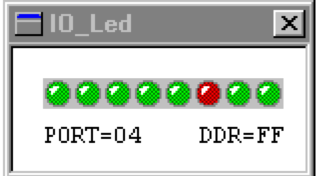


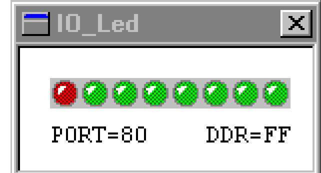
8. Choose *Simulator | Load* and load `FIBO.ABS`.
9. Click on the *Stimulation* window to activate it.
10. Choose *Stimulation* | *Execute*.



11. Choose *Run | Start/Continue*.
12. As soon as execution of the application is started, the simulator computes the number of execution cycles. Once the number of cycles associated with a command in the stimulation file are reached, the corresponding action is taken.

The *IO_Led* component contains the digital representation of the memory location 0x210. So any change performed on the object ‘a’ is displayed directly in the *IO_Led* Component. During execution of the application, the *IO_Led* component periodically switches between the two following LED status:





Stimulated Interrupt

Using an editor, open following stimulation file `IO_INT.TXT` in directory `C:\HICROSS\DEMO\SIM68KC`.

```
def a = TargetObject.#210.B;

PERIODICAL 200000, 10:
  100000 RAISE 7, 3, "test_interrupt";
END

10000000 RAISE 7, 3, "test_interrupt";
```

In the first line, the stimulated object is defined. This object is located at address 0x210 and is 1 byte wide.

Once 200'000 cycles have been executed, the interrupt is raised periodically 10 times in line 3. The RAISE command takes the number of the interrupt in the interrupt vector map as first argument. This number, "7" in our example, refers to the Real Time Interrupt for the M68K.

To export this example to a different target, take a look at the interrupt vector map in the technical data manual of the matching MCU.

The second argument specifies the interrupt priority and the third argument is a free chosen name of the interrupt.

1. Quit the editor without saving `IO_INT.TXT` and start *HI-WAVE*.
2. Load the *IO_LED* and the *Stimulat* components.
3. Click on the *Stimulation* window to activate it.
4. Choose *Stimulation* | *Open File* and open `IO_INT.TXT`.
5. Choose *Simulator* / *Load* and load `IO_STIMU.ABS`.
6. Select the *Source* component window.
7. Choose *Source* | *Open Module* `IO_STIMU.C`.
8. Scroll to the procedure *RTI_interrupt*.
9. Set a breakpoint into the *RTI_interrupt* function source.
10. Click on the *Stimulation* window to activate it.
11. Choose *Stimulation* | *Execute*.
12. Choose *Run* / *Start/Continue*.

13. The execution of the application is stopped at the breakpoint in the interrupt procedure. The number of CPU Cycle is about 300000.
14. Choose *Run | Start/Continue*. The breakpoint in the interrupt procedure is once more reached,   the number of CPU Cycle is about 400000.
15. Choose *Run | Start/Continue.*

The file `IO_INT.TXT` is used to generate 11 interrupts. 10 periodical interrupts are generated every 100000 CPU cycles from 200000+100000=300000 to 1200000 CPU cycles. A last one is generated when the number of CPU cycles reaches 10000000 of cycles.

© Copyright 1998 HIWARE

How to Improve Code Efficiency

Introduction

This section gives you useful information and hints to improve the code efficiency.

The Compiler tries whenever possible to generate compact and efficient code. But not everything can be handled directly by the Compiler, and with a little help from the programmer it is possible to reach denser code. Some Compiler options or usage of SHORT segments (if available) can help you to generate compact code with the Compiler.

Compiler Options

Usage of following compiler options will help you to reduce the size of the code generated. Note that not all options may be available for your target.

-Or: Register Optimization

When accessing pointer fields, this option will prevent the compiler to reload the address of the pointer for each access. An index register may hold the pointer value over statements if possible.

-Cni: Non integral promotion on integer

When this option is specified, the ANSI C integral promotion does not apply to character comparison or arithmetic. This can save a big amount of code.

-Oc: Common Subexpression Elimination

The compiler tries to perform CSE: instead evaluating the same expression several times, the Compiler evaluates it once and stores it on the stack for later retrieval.

SHORT Segments

Variables allocated on the direct page (between 0 and 0xFF) can be accessed using the direct addressing mode. You can tell the Compiler that some variables are allocated on the direct page, if you define them in a SHORT segment (not available for

all targets).

Example:

```
#pragma DATA_SEG SHORT myShortSegment
unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
unsigned int myvar3, myVar4.
```

In the previous example, ‘myVar1’ and ‘myVar2’ are both accessed using the direct addressing mode. Variables ‘myVar3’ and ‘myVar4’ are accessed using the extended addressing mode.

When some exported variables are defined in a SHORT segment, the external declaration for these variables must also specify that they are allocated in a SHORT segment. The External definition of the variable defined above will look like:

```
#pragma DATA_SEG SHORT myShortSegment
extern unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
extern unsigned int myvar3, myVar4.
```

Then the segment must be placed on the direct page in the PRM file.

Example:

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM                    INTO  MY_ROM;
    DEFAULT_RAM                    INTO  MY_RAM;
    _DATA_ZEROPAGE, myShortSegment INTO  Z_RAM;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

Be careful: The linker is case sensitive. The segment name must be identical in the C and PRM file.

Defining IO Registers

IO Registers are usually base at address 0. In order to tell the compiler it should use direct addressing mode to access the IO registers, these registers can be defined in a SHORT section (if available) based at the specified address.

In the C source file, the IO register are defined as follows:

Example:

© Copyright 1998 HIWARE

```
typedef struct {
   unsigned char SCC1;
   unsigned char SCC2;
   unsigned char SCC3;
   unsigned char SCS1;
   unsigned char SCS2;
   unsigned char SCD;
   unsigned char SCBR;
} SCIStruct;
#pragma DATA_SEG SHORT SCIRegs
SCIStruct   SCI;
#pragma DATA_SEG DEFAULT
```

Then the segment must be placed at the appropriate address in the PRM file.

Example:

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    SCI_RG = READ_WRITE  0x0013 TO 0x0019;
    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM                    INTO  MY_ROM;
    DEFAULT_RAM                    INTO  MY_RAM;
    _DATA_ZEROPAGE                 INTO  Z_RAM;
    SCIRegs                        INTO  SCI_RG;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

Be careful: The linker is case sensitive. The segment name must be identical in the C and PRM file.

Programming Guide Lines

Following few programming guidelines can also reduce the code size. Many things are optimized by the Compiler, but if the programming style is very complex or if it forces the Compiler to perform special code sequences, code efficiency is maybe not what you expect.

HLI Assembly

If possible, do not mix High Level Inline (HLI) Assembly. Using HLI assembly may affect the register trace of the compiler, because the Compiler cannot touch HLI Assembly and thus it is out of range for any optimizations (except branch optimization, of course). The Compiler in the worst case has to assume that everything has changed, e.g. it cannot hold variables into registers over HLI statements. Nor-

mally it is better to place special HLI code sequences into separate functions, but then there is the drawback of an additional call/return. But placing HLI instructions into separate functions (and module) will simplify porting the software to another target too.

Example (not recommended):

```
void foo(void) {
  /* some local variable declarations */
  /* some C/C++ statements */
  asm {
    /* some HLI statments */
  }
  /* maybe other C/C++ statements */
}
```

Example (recommended):

```
/* hardware.c */
void special_hli(void) {
  asm {
    /* some HLI statments */
  }
}

/* foo.c */
void foo(void) {
  /* some local variable declarations */
  /* some C/C++ statements */
  special_hli();
  /* maybe other C/C++ statements */
}
```

Post/Pre Operators in Complex Expressions

Writing a program complex may also result into complex code. In general it is the job of the compiler to optimize also very complex functions, but some rules may help the compiler to generate efficient code.

If your target does not support powerful postincrement/postdecrement/preIncrement/predecrement instructions, it is not recommended to use the '++' and '--' operator in complex expressions. Especially postinc/dec may result additional code:

```
a[i++] = b[--j];
```

Better write above statement as

```
i--; a[i] = b[j]; j--;
```

Using it in simple expressions as

```
i++;
```

is not a problem, of course.

Additionally, avoid assignments in parameter passing or side effects (as ‘++’ and ‘--’). The problem is that evaluation order of parameters is undefined (ANSI C standard 6.3.2.2) and may vary from Compiler to Compiler and even from one release to another:

```
i = 3;
foo(i++, --i);
```

In the above example, foo() will be called either with ‘foo(3,3)’ or with ‘foo(2,2)’.

Boolean Types

In C, the boolean type of an expression is an ‘int’. A variable or expression evaluating to ‘0’ (zero) is FALSE and everything else (!= 0) is TRUE. Instead using an ‘int’ (usually 16 or 32 bit), it may be better to use an 8bit type to hold a boolean result. For ANSI-C compliance, the basic boolean types are declared in ‘stdtypes.h’:

```
typedef int Bool;
#define TRUE  1
#define FALSE 0
```

Using

```
typedef Byte Bool_8;
```

from ‘stdtypes.h’ (‘Byte’ is an unsigned 8bit data type also declared in ‘stdtypes.h’) reduces memory usage and improves code density.

Bitfields

Using bitfields to save memory may be a bad idea, because often bitfields has to produce a lot of additional code. For ANSI-C compliance, bitfields do have a type of ‘signed int’, thus a bitfield of size 1 is either ‘-1’ or ‘0’, this could force the compiler to ‘sign extend’ operations:

```
struct {
  int b:0; /* -1 or 0 */
} B;
int i = B.b; /* load the bit, sign extend it to -1 or 0 */
```

Sign extensions are normally time and code inefficient operations.

Struct Returns

ANSI-C/C++ allows functions returning a struct. Except if the struct can be

returned in a single register, returning a struct may force the Compiler to produce a lot of code:

```
struct S foo(void) {
  /* ... */
  return s; // (4)
}

void main(void) {
  struct S s;
  /* ... */
  s = foo();  // (1), (2), (3)
  /* ... */
}
```

Normally the compiler has first to allocate space on the stack for the return value (1) and then to call the function (2). In the callee ‘foo’ during the return sequence, the Compiler has to copy the return value (4, struct copy). Depending on the size of the struct this may be done inline or with a special memory copy routine (strcpy). After return, the caller ‘main’ has to copy the result back into ‘s’. Depending on the Compiler/Target, it is possible to optimize some sequences, e.g. avoiding some copy operations, but in general it is quite more complex than it looks like. Returning a struct by value may use a lot of execution time, means a lot of code and stack usage.

A better way is to pass only a pointer to the callee for the return value:

```
void foo(struct S *sp) {
  /* ... */
  *sp = s; // (4)
}

void main(void) {
  S s;
  /* ... */
  foo(&s);  // (2)
  /* ... */
}
```

With the above example, the Compiler just has to pass the destination address and to call ‘foo’ (2). On the callee side, the callee copies the result indirect into the destination (4). This approach reduces stack usage, avoids to copy structs and results in denser code. Note that the Compiler may also the above sequence (if supported), but for rare cases the above sequence is not exactly the same as returning the struct by value (e.g. if the destination struct is modified in the callee).

Local Variables

Using local variables instead global ones results into better maintainability of the application, because side effects are reduced or totally avoided. Using local variables/parameters reduces global memory usage, but increases stack usage. Thus

stack access capabilites of the target influences the code quality. Depending on the target capabilities access to local variables may be very inefficient. Reasons are: no dedicated stack pointer (another address register has to be used instead, thus it might not be used for other values) or access to local variables is inefficient due the target architecture (limited offsets, only few addressing modes). Additionally allocating a huge amount of local variables may be inefficient because the Compiler has to generate a complex sequence to allocate the stack frame in the beginning of the function and to deallocate them in the exit part:

```
void foo(void) {
  /* huge amount of local variables: allocate space! */
  /* ... */
  /* deallocate huge amount of local variables */
}
```

If the Target provides special entry/exit instructions for such cases, allocation of many local variables is not a problem. A solution is to use global or static local variables, but this deteriorates maintainability and also may waste global address space. As a solution the Compiler may offer an option to overlap parameter/local variables using a technique called ‘overlapping’: Local variables/parameters are allocated as global ones, and the linker overlaps them depending on their usage. For targets with limited stack (e.g. no stack addressing capabilities) this often is the only solution, but with this solution the code is not reentrant (no recursion is allowed).

Parameter Passing

Avoid parameters which exceed the data passed through registers (see Back End).

Unsigned Data Types

Using unsigned data types is normally ok because signed operations are normally much more complex than unsigned ones (e.g. shifts, divisions and bitfield operations). But normally it is a bad idea to use unsigned types just because a value is always larger or equal to zero and because the type can hold a larger positive number.

Inlining/Macros

abs() and absl()

Instead calling abs() and absl() in the stdlib, it is better to use the corresponding macro M_ABS defined in stdlib.h:

```
/* extract
/* macro definitions of abs() and labs() */
#define M_ABS(j)  (((j) >= 0) ? (j) : -(j))
```

```
extern int      abs   (int j);
extern long int labs  (long int j);
```

But be careful: because M_ABS() is a macro,

```
i = M_ABS(j++);
```

is not the same as

```
i = abs(j++);
```

memcpy() and memcpy2()

ANSI-C requires that the memcpy() library function in ‘strings.h’ returns a pointer of the destination and handles and is able to handle also a count of zero:

```
/* extract of string.h */
extern void *memcpy(void *dest, const void *source, size_t count);
extern void memcpy2(void *dest, const void* source, size_t count);
  /* this function does not return the dest and assumes count > 0 */

/* extract of string.c */
void *memcpy(void *dest, const void *source, size_t count) {
  uchar *sd = dest;
  uchar *ss = source;
  while (count--)
     *sd++ = *ss++;
  return (dest);
}
```

But if the function does not have to return the destination and it does have to handle a count of zero, the memcpy below is much more simpler and faster:

```
/* extract of string.c */
void memcpy2(void *dest, const void* source, size_t count) {
  /* this function does not return the dest and assumes count > 0 */
  do {
    *((uchar *)dest)++ = *((uchar*)source)++;
  } while(count--);
}
```

Thus replacing calls to memcpy() with calls to memcpy2() may save a lot of runtime and code size.

Short Segments

Whenever possible and available (not all targets support it), place frequently used global variables into a DIRECT or SHORT segment using

```
#pragma DATA_SEG SHORT MySeg
```